# Compass User Manual:
# A Tool for Source Code Checking
# (A ROSE Tool)
# Draft User Manual
# (Associated with ROSE Version 0.9.3a)

**ROSE Team**
Lawrence Livermore National Laboratory
Livermore, CA 94550
925-423-2668 (office) 925-422-6278 (fax)
dquinlan@llnl.gov
Project Web Page: http://www.rosecompiler.org
UCRL Number for ROSE User Manual: UCRL-SM-210137-DRAFT
UCRL Number for ROSE Tutorial: UCRL-SM-210032-DRAFT
UCRL Number for ROSE Source Code: UCRL-CODE-155962

ROSE User Manual (pdf)
ROSE Tutorial (pdf)
ROSE HTML Reference (html only)

September 23, 2008

# Contents


**1 Introduction** **5**
- 1.1 Overview . . . 5

**2 Design and Verification** **7**
- 2.1 Usage Model . . . 7
- 2.2 Trust Model . . . 7
- 2.3 Architecture . . . 9
- 2.4 Design . . . 10
- 2.5 Compass Verifier . . . 11
- 2.6 Future Work . . . 13

**3 Using Compass** **15**
- 3.1 Running Compass . . . 15
- 3.2 Output from Compass . . . 15
- 3.3 How To Write A New Checker . . . 24
- 3.4 Including/Excluding Checkers in the Compass Build Process . . . 25
- 3.5 Including/Excluding Checkers During Compass Execution . . . 26
- 3.6 Including/Excluding Paths and Filenames with Compass . . . 26
- 3.7 Checking Security Properties of Checkers . . . 26
- 3.8 Testing Compass and its Checkers . . . 27
- 3.9 Extending the Compass Infrastructure . . . 27

**4 Using Compass Verifier** **31**

**5 Compass Checkers** **33**
- 5.1 Allocate And Free Memory In The Same Module At The Same Level Of Abstraction . . . 34
- 5.2 Allowed Functions . . . 37
- 5.3 Assignment Operator Check Self . . . 39
- 5.4 Assignment Return Const This . . . 41
- 5.5 Avoid Using The Same Handler For Multiple Signals . . . 43
- 5.6 Binary Buffer Overflow . . . 46
- 5.7 Binary Interrupt Analysis . . . 47
- 5.8 Bin Print Asm Functions . . . 48
- 5.9 Bin Print Asm Instruction . . . 49
- 5.10 Boolean Is Has . . . 50
- 5.11 [No Reference] Buffer Overflow Functions . . . 52

5.12 Secure Coding : EXP04-A. Do not perform byte-by-byte comparisons between structures . . . 54
5.13 Char Star For String . . . 56
5.14 Comma Operator . . . 58
5.15 [No Reference] Computational Functions . . . 59
5.16 Const Cast . . . 60
5.17 Constructor Destructor Calls Virtual Function . . . 61
5.18 Secure Coding : STR05-A. Prefer making string literals const-qualified . . . 63
5.19 Control Variable Test Against Function . . . 65
5.20 Copy Constructor Const Arg . . . 67
5.21 Cpp Calls Setjmp Longjmp . . . 69
5.22 Cycle Detection . . . 71
5.23 Paper: Cyclomatic Complexity . . . 72
5.24 Data Member Access . . . 73
5.25 Deep Nesting . . . 75
5.26 Default Case . . . 77
5.27 Default Constructor . . . 79
5.28 Discard Assignment . . . 81
5.29 Do Not Call Putenv With Auto Var . . . 82
5.30 Do Not Delete This . . . 84
5.31 Do Not Use C-style Casts . . . 86
5.32 Duffs Device . . . 88
5.33 Dynamic Cast . . . 90
5.34 Empty Instead Of Size . . . 92
5.35 Enum Declaration Namespace Class Scope . . . 94
5.36 CERT-DCL04-A: Explicit Char Sign . . . 96
5.37 Explicit Copy . . . 98
5.38 Explicit Test For Non Boolean Value . . . 99
5.39 File Read Only Access . . . 101
5.40 Float For Loop Counter . . . 103
5.41 Floating Point Exact Comparison . . . 104
5.42 Fopen Format Parameter . . . 106
5.43 Forbidden Functions . . . 108
5.44 For Loop Construction Control Stmt . . . 111
5.45 For Loop Cpp Index Variable Declaration . . . 113
5.46 Friend Declaration Modifier . . . 115
5.47 Function Call Allocates Multiple Resources . . . 117
5.48 CERT-DCL31-C: Function Definition Prototype . . . 119
5.49 Paper: Function Documentation . . . 121
5.50 Induction Variable Update . . . 122
5.51 Internal Data Sharing . . . 124
5.52 Localized Variables . . . 126
5.53 [No Reference] : Loc Per Function . . . 128
5.54 Lower Range Limit . . . 130
5.55 Magic Number . . . 132
5.56 Malloc Return Value Used In If Stmt . . . 134
5.57 Multiple Public Inheritance . . . 136
5.58 Name All Parameters . . . 138

5.59 [No Reference] : New Delete . . . 139
5.60 No Asm Stmts Ops . . . 141
5.61 No Exceptions . . . 142
5.62 No Exit In Mpi Code . . . 143
5.63 No Goto . . . 145
5.64 Non Associative Relational Operators . . . 146
5.65 Nonmember Function Interface Namespace . . . 148
5.66 Non Standard Type Ref Args . . . 150
5.67 Non Standard Type Ref Returns . . . 152
5.68 Non Virtual Redefinition . . . 154
5.69 No Overload Ampersand . . . 156
5.70 CERT-MSC30-C: No Rand . . . 158
5.71 No Second Term Side Effects . . . 160
5.72 Secure Coding : EXP06-A. Operands to the sizeof operator should not contain side effects . . . 162
5.73 No Template Usage . . . 164
5.74 No Variadic Functions . . . 166
5.75 CERT-POS33-C: No Vfork . . . 168
5.76 CERT EXP33-C and EXP34-C : Null Dereference . . . 170
5.77 CERT-DCL04-A: One Line Per Declaration . . . 174
5.78 Operator Overloading . . . 176
5.79 Other Argument . . . 178
5.80 Place Constant On The Lhs . . . 179
5.81 Prefer Algorithms . . . 181
5.82 Secure Coding : FIO07-A. Prefer fseek() to rewind() . . . 183
5.83 Prefer Setvbuf To Setbuf . . . 185
5.84 Protect Virtual Methods . . . 187
5.85 Push Back . . . 189
5.86 Right Shift Mask . . . 192
5.87 Set Pointers To Null . . . 194
5.88 Single Parameter Constructor Explicit Modifier . . . 196
5.89 Size Of Pointer . . . 198
5.90 Secure Coding : INT06-A. Use strtol() to convert a string token to an integer . . . 200
5.91 Sub Expression Evaluation Order . . . 202
5.92 Ternary Operator . . . 204
5.93 Time_t Direct Manipulation . . . 206
5.94 Unary Minus . . . 208
5.95 Uninitialized Definition . . . 209
5.96 Upper Range Limit . . . 211
5.97 Variable Name Equals Database Name . . . 213
5.98 Void Star . . . 215

**6 Appendix 217**
6.1 Design And Extensibility of Compass Detectors . . . 217

## Acknowledgments

This tool is the product of the entire ROSE team. Compass depends upon the ROSE open compiler infrastructure and is a simple application of mechanisms in ROSE which have been developed over several years and contributed to by a large number of people.

This currently includes:

- Staff: Dan Quinlan
- Post docs: Thomas Panas, Chunhua Liao, and Jeremiah Willcock
- Ex Post doc: Richard Vuduc
- Students:
  Gergo Barany, Michael Byrd, Valentin David, Han Kim, Robert Preissl, Andreas Saebjornsen, Jacob Sorensen, Ramakrishna Upadrasta, Jeremiah Willcock, Gary Yuan
- Internal LLNL Contributors: Greg White
- External Collaborators:
  We want to thank CERT, who has been particularly helpful and supportive both with the development of checkers for their Secure Coding Rules and with numerous suggestions.

# Chapter 1

# Introduction

## 1.1 Overview

Compass is a tool for the checking of source code. It is based on the ROSE compiler infrastructure and demonstrates to use of ROSE to build lots of simple pattern detectors for analysis of C, C++, and Fortran source code.

The purpose of this work is several fold:

- Provide a concrete tool to support interactions with lab customers.
- Provide a home for the security analysis specific detectors being built within external research projects.
- Provide an external tool for general analysis of software.
- Provide a tool to support improvements to the ROSE source code base.
- Define an infrastructure for an evolving and easily tailored program analysis tool.
- Provide a simple motivation for expanded use of ROSE by external users. Development, testing, and evaluation of ROSE infrastructure is best facilitated through its expanded use by others and this provides a specific and attractive tool that can provide feedback to users about their own code projects. Even though optimization research is our focus, this gets our supporting infrastructure for optimization research out and into use by others in the form of an extensible tool.

Note that as the collection of detectors grows we will periodically reorganize the collection. At some point soon we will build a hierarchy to organize the evolving collection.

**A basis for other source analysis tools** Input and output to ROSE is organized so that any number of sources could be used. So although we provide a compiler interface (for simplicity), we will also provide a GUI interface as an alternative interface to demonstrate that the detectors are orthogonal to there use in alternative tools. Alternative tool interfaces should be possible and will

further demonstrate the independence of the input and output mechanisms to the designs and implementation of the core detectors.

**Add Your Own Detector** Detectors written in Compass make direct use of ROSE and are designed to be copied and extended by users to develop their own detectors. We welcome the contribution of these detectors back to the ROSE team for inclusion into future releases of Compass; full credit for all work will be provide to all authors. Compass is an open source project using ROSE, an open source compiler infrastructure.

Each of the detectors are examples of how to add your own detector to **Compass**. If you build a detector that you would like to have be distributed with **Compass**, please send it to us and we will add your as an external contributor.

Guidelines for contributions:

- Use any Compass detector and an example.
- provide the documentation about your detector.
- Use any features in ROSE to support your detector; AST, Control Flow graph, System dependence Graph, Call Graph, Class Hierarchy Graph, etc.
- Your detector should have **NO** side-effects on the AST.

# Chapter 2

# Design and Verification

Compass is a tool is used to analyze software (both source code and binaries). A collection of *checkers* are built with each of them detecting the violation of a rule. By reporting on the violations of rules *Compass* provides a way to enforce predefined or arbitrary user specified properties on software. This chapter covers the design of *Compass* and the design of the verification in the *Compass Verifier*, used to verify properties of the checkers implemented and submitted to *Compass*.

## 2.1 Usage Model

Figure 2.1 shows the usage model (use cases) of *Compass*. The analysis is triggered by the user running *Compass* over an input file (source code or binary). The user implicitly selects which checkers to execute (defining what rules are to be enforced); by default all checkers are run. The user also specifies the input file to be checked; for source code the specification is similar to the command line required for the compilation in the case of a source file. Results of the analysis are presented to the user, a number of mechanisms can be used to display the results.

Either the same user or a different user/developer can also implement and submit checkers to be built into Compass. Since external users may contribute checker automatically via scripts, a verification of the validity and safety of these checkers is necessary. We provide a *Compass Verifier* that helps to check that all checkers are safe. Currently, the verifier is run by an administrative person but may run automatically in the future.

## 2.2 Trust Model

By design we make a few assumptions about the use of Compass in order to define a secure tool. We assume:

1. For now, there is an assumption of trust in the person writing the checker. We use the *Compass Verifier* as a way to double check the checkers so that we can eventually weaken the level of trust assumed for people writing checkers. However, the design of the *Compass Verifier* is not likely to

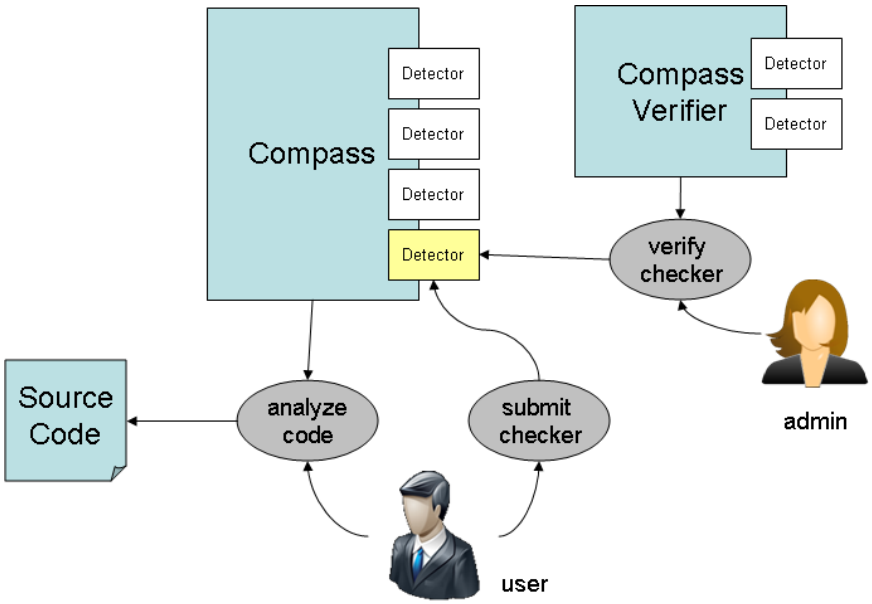


Figure 2.1: Compass Use Case

ever be robust enough to guarantee an automated proof of security for each checker. Thus, we also assume that someone trusted will also review the checker. *not implemented: We expect that a digital signature (using a key mechanism) is possible to associate a trusted reviewer with a review of the checker together with a strong hash function that digitally signs the checker source code.*

2. Since running *Compass Verifier* is an optional part of building the Compass executable, the person running these test is trusted. There are two ways to run the *Compass Verifier* (see section 3.7 for details):

   - Slow: once on each checker (`make verify`). This mechanism tests all the files one checker at a time and thus can not miss a file. Note that even the counter examples are tests which can be a problem when the counter example for the checker is detected as a violation for *Compass Verifier*. Counter examples for checkers have to be carefully written to not represent examples that violate *Compass Verifier*.
   - Fast: once on the union of all the checkers (`make oneBigVerify`). This step forms a single file of all the checkers (and in-doing so can miss some files, and so is less secure). It is mostly for testing purposes.

3. The person building the Compass executable is trusted.

4. The environment where the testing using Compass is done is trusted.

5. Compass is designed so that the user running Compass need not be trusted.

It is unclear at present how weak the assumption of trust on the compass checker developer can be and it may ultimately depend directly on the capabilities of the *Compass Verifier*.

## 2.3 Architecture

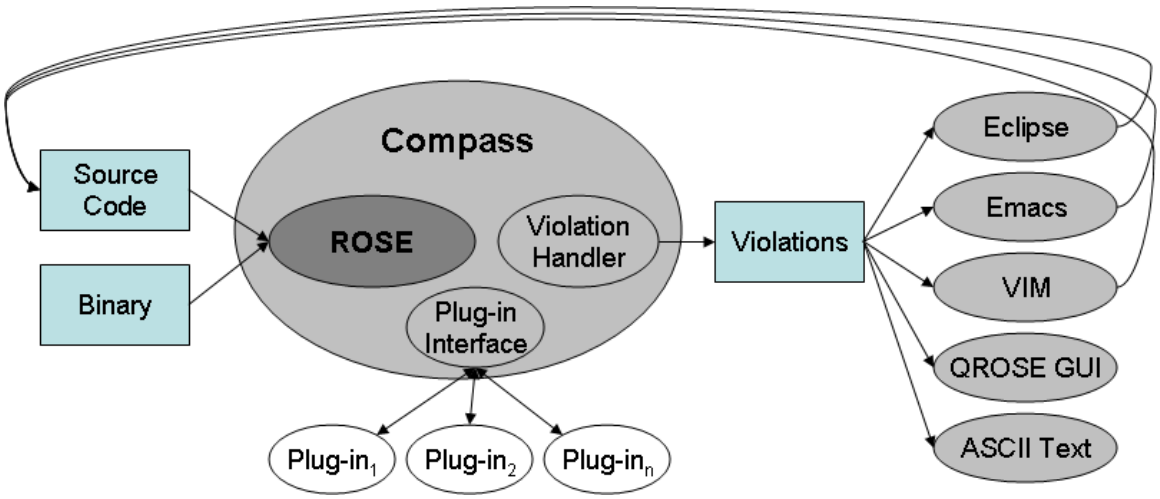


Figure 2.2: Compass Architecture

Compass is a tool that allows users to implement checkers to locate and report software defects. Documentation of various kinds of software defects can be found in sources such as the CERT Secure Coding Rules, Common Weakness Enumeration from MITRE, and other sources. Our focus is not to define new software defects but rather to provide a platform that allows the easy implementation of defect checkers. Compass has been designed to be easy to extend, allowing users to implement their own custom checkers (custom source code analyses for identifying defects), as shown in Figure 2.2. Compass supports the implementation of both simple as well as more advanced defect checkers. For the latter, Compass utilizes the ROSE infrastructure to perform a wide range of general purpose program analyses, such as control flow analysis, data flow analysis, program slicing, etc.

Compass is designed in a way that allows users who do not necessarily have compiler backgrounds to utilize the ROSE infrastructure to build their own analysis tools. Compass is foremost an extensible open source infrastructure for the development of large collections of rules. Our current implementation supports automatic defect checking, programming language restriction, and malware detection in C, C++, and object code. Support for Fortran is a new addition to ROSE and will be supported in Compass in the near future.

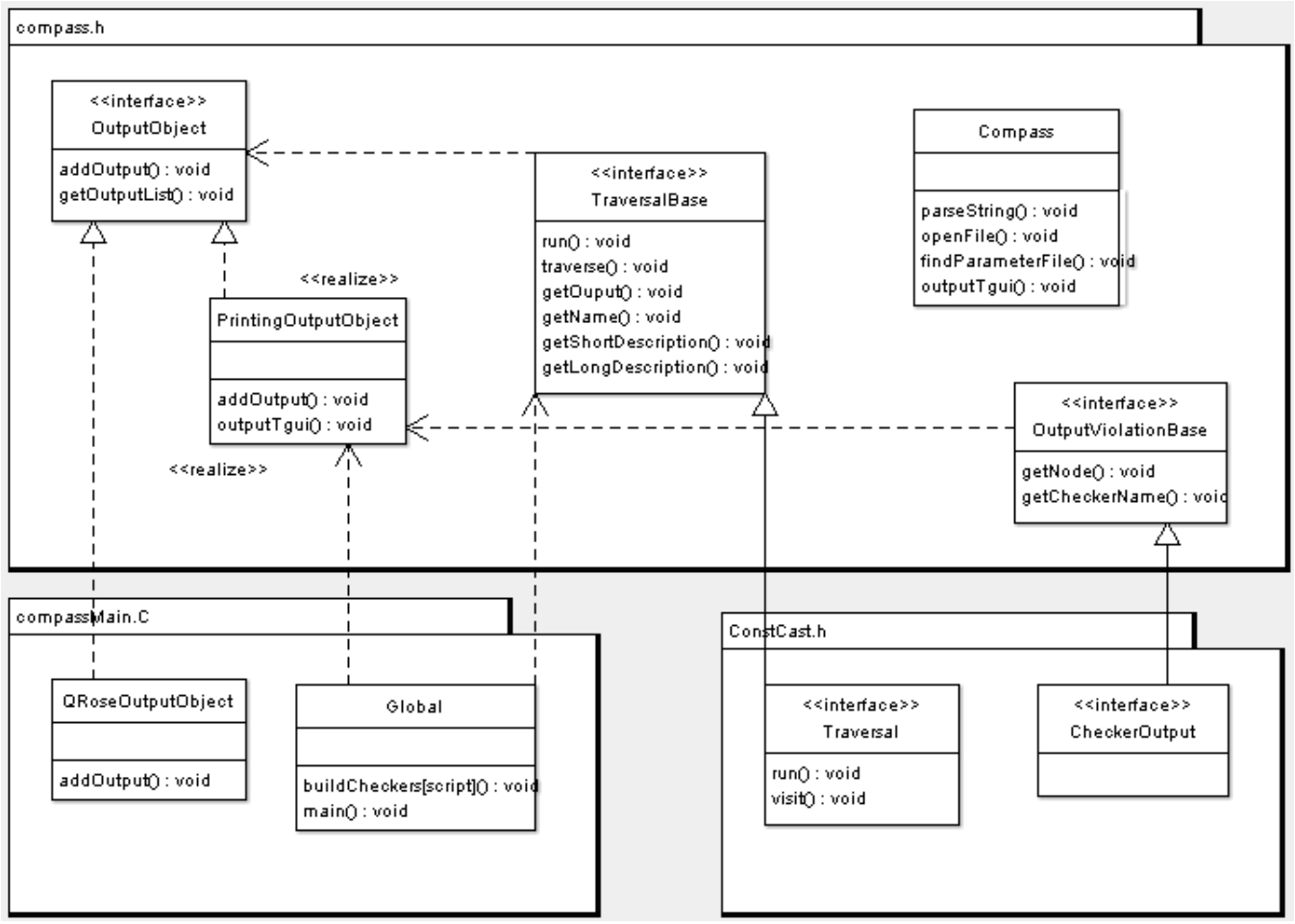


Figure 2.3: Compass Design

## 2.4 Design

Compass is designed to be easy to extend. Any user may write a checker and add it to Compass. Figure 2.3 illustrates the UML design decisions behind Compass.

Most of the functionality of Compass is in abstract classes hidden in the Compass namespace within compass.h - a file within the compassSupport directory. The figure uses a specific example, *ConstCast*, for illustration; Compass is designed to support a large number of checkers (hundreds). All checkers, such as the *ConstCast* checker (illustrated in figure 2.3), utilize the abstract classes to traverse a program with all its nodes and to output violations found in that code according to the local algorithm.

CompassMain is the main executable that initially calls ROSE to parse a program. Then *buildCheckers* is called to load all checkers that are specified within a configuration file. The configuration file allows users to turn on and

turn off specific checkers for their run-time analyses. However, the configuration file only permits checkers to be loaded that were part of Compass at compile-time.

The main interface file compass.h contains the class *Checker* and the abstract class *OutputObject*. Checker is the interface to ROSE, giving the metadata for a checker and a function to apply the checker to a given AST. OutputObject aids to output defects found by a specific checker. More functionality to handle e.g. file input and parameters provided to Compass, is provided within the Compass namespace.

## 2.5 Compass Verifier

Compass must be safe, so that analyses and their results can be trusted. The *Compass Verifier* is used at build time to run a specific set of separate compass rules offer the source code of all the checkers. For simplicity it runs in two modes: fast, for checking specific named checker source files; and slow, for testing *all* checker source files.

### 2.5.1 Threats

In order to define a complete design for security we outline the threats that understand to be relevant. The main threats to the validity of Compass checkers are:

- *Malicious User*
  A malicious user is an external user of Compass contributing a checker that performs malicious behavior.

- *Malicious Checker*
  Compass is extensible and new checkers can be added externally (users outside the main development group). A checker can be programmed arbitrarily using the C/C++ and assembly programming languages. It is therefore possible for a skilled programmer to hide malicious operations within a checker. Compass must prevent checkers with malicious behavior to be part of the Compass system. Threats are:

  - **exfiltration**
    A checker should act in a secure way with the input files it is given. Securing the inputs to Compass (e.g. the inputs to each checker) from exfiltation is a first priority. Allowing a checker to scan the host machine to exfiltrate arbitrary data (this is a threat that any secure software will have).

  - **modification of filesystem**
    A checker should be side-effect free, or have only well defined side-effects, but a malicious checker could modify or erase parts of the accessible file system (e.g. deleting whole directory structures).

- *Malicious Compass*
  Since Compass is built from ROSE, it is possible to modify compass (or

any checker) to generate source code that could be compiled to replace the existing executable (there are some constraints here) or regenerate the source code to replace the existing source code or perhaps just provide an alternative copy of the source code. This indirect transformation of the input code is a threat.

- *Source Code Replacement*
  It should not be possible for users to exchange the source code of checkers within a running system, i.e. Compass cannot implement dynamic loading of checkers. Such a feature would compromise its safety.

- *Binary Replacement*
  Another threat is the replacement of a valid Compass checker with a modified malicious version within a binary release of Compass. Therefore, Compass should be aware if parts of itself were modified and should not execute.

### 2.5.2 Mitigation of Threats

Compass is designed to be safe. The Compass Verifier is a stable separate copy of Compass that contains only a few checkers to check (external and internal) user delivered checkers for safety. We have hopefully designed Compass in a way that it addresses the threats mentioned above:

- *Malicious User*
  Initially, we permit only trusted individuals to add new checkers to Compass. Once the verification process is matured, we will extend this policy to allow less trusted users to contribute to Compass. A goal will be to allow arbitrary users to contribute checkers, however, a review of the whole Compass design (and the *Compass Verifier* especially) will be required to define required trust levels for user/developers who implement checkers.

**FIXME:** *We might define trusted and untrusted checkers as a way to have checkers from arbitrary users, but mark them as untrusted.*

- *Malicious Checker*
  To prevent Compass from executing malicious code, the Compass Verifier executes its own checkers on any user defined checker that is being considered to be added to Compass. Currently, the Compass Verifier contains three checkers:

  - *fileReadOnlyAccess* ensures that a user defined checker performs no write or execute operations on files.
  - *allowedFunctions* is a *white list* of function calls permitted in a checker. This list contains functions that are trusted and hence considered safe when integrated to Compass.

**FIXME:** *This is not yet implemented as a white list and is instead currently a black list; called: forbiddenFunctions.*

  - *noAsmStmtsOps* searches for assembly instructions in a checker and flags and reports all cases as unsafe.
  - To avoid modifications of the AST for the purpose of allowing other checkers to pass, the AST should not be modified (this should extend to all the program analysis graphs generated and used by other checkers). *This is not implemented yet.*

- *Malicious Compass*
  Since Compass does not generate code, it can not be used to modify the input software (source code or binary) or generate a new copy that could be confused with the input. However, future versions of Compass make make transformation to introduce greater levels of security; fix flaws, mitigate specific forms of threats, etc. It will be important to make sure that such transformation can not change the behavior of an input code to make the modified input code malicious. Current proposed approaches would build a patch which would have to be inspected by a trusted developer before it would be applied to modify the input code.

- *Source Code Replacement*
  Checkers can only be added at compile time to Compass, not at run-time. This means that checkers (meaning the source code) cannot be exchanged against unsafe versions at run-time. Furthermore, we allow only the Compass tool builder (admin) to build versions of Compass that must pass the Compass Verifier.

- *Binary Replacement*
  Our goal is to perform a strong hash (e.g. Secure Hash Algorithm - SHA2) as a checksum on all the checkers part of the binary Compass distribution before Compass is executed. In this way Compass will not run if parts of it were modified. *This is not implemented yet.*

**FIXME:** *We should describe the policy for allowing SHA2 to be verified. Where the SHA2 results for checkers would be published (e.g. web site), etc.*

## 2.6 Future Work

Currently we are engaged in design reviews with CERT, we expect that this will lead to improvements in the security to support a key based approach to a trusted execution of tools built within the compass infrastructure, including Compass itself.

# Chapter 3

# Using Compass

## 3.1 Running Compass

Compass is currently distributed as part of ROSE, and represents one of many tools that can be built using the ROSE open compiler infrastructure. Compass resides in the `ROSE/projects/compass`. The compass project is currently divided into three subdirectories representing the compass instrastructure, extensions (checkers), and individual compass-like tools. As part of building ROSE Compass will be automatically built in the compass directory. Running compass is a matter if typing `compassMain` and handing in a number of options. The `compassMain` program acts just like a compiler so it is appropriate to hand it the same options required to compile your source file (e.g `-I` directory paths and a source file. Compass will figure out the language from the source file suffix. Using the `--help` option will provide a more complete list of options available to ROSE based tools. See also the section of this chapter on the include/exclude options for path and file names as these will permit the output from header files to be tailored.

## 3.2 Output from Compass

Output from compass can be generated in a number of forms, the default is ASCII text output of the messages about rule violations with the source code position in *GNU standard source code position format*. This form can be used to interact with external tools (e.g. Emacs) to permit alternative interface to Compass. Mechanisms available include:

- Emacs:
  Detecting errors while you type 3.5 and 3.1.
- Vim 7:
  Compass can work with Vim 7's QuickFix commands to highlight source lines with error messages 3.9.
- CompassGUI:
  There is also a Compass GUI for reviewing Compass output and interactively rerunning compass and sifting through the output while relating

them to the source code 3.2. This work uses the QRose library produced at Imperial College London by Gabriel Coutinho, as part of their development of FPGA tools using ROSE. QRose is based on the Qt library and provides a wide number of ROSE aware components to make the development of GUIs for ROSE based tools easy. The source code for the Compass GUI is provided, but this work is unfinished (and required the QRose library available from Imperial).

- ToolGear post-processing:
  Output in XML permits the use of ToolGear (LLNL tool available on the web) for viewing Compass generated output. This mechanism is particularly useful for reviewing the results of nightly builds (and associated runs of large projects using Compass). See figure 3.3

- ASCII output:
  Output in ASCII format is of the form shown in 3.4. This form permits the connection to multiple external tools (the Emacs interface reads the ASCII output format directly).

### 3.2.1 Using Compass With Emacs

Compass as a checker is most useful when the user is notified as early as possible when he violates a desired software property. Although for many purposes it is sufficient to run Compass separately; it is possible to use compass seamlessly when developing in emacs. By using an emacs extension called *flymake* together with Compass erroneous lines can be highlighted while programming, and the relevant error messages displayed in a dialog. Syntax errors from ROSE will be displayed as well, see figure 3.1.

Much thanks for David Svoboda at CERT at CMU for first configuring Flymake to work with Compass and demonstrating the idea. It has provided a great way to check code using compass and its use in Emacs has stimulate a number of ideas that have made their way back into Compass.

#### Emacs .emacs Code Requirement

Figure 3.5 shows the code that is required to be added to the `.emacs` file. A copy of this code is available in the compass source directory in the file `emacs_compass_config.el`.

#### Emacs Version Requirement

Emacs version 22 or newer is required to take advantage of the emacs integration of Compass. Before using Compass a 3 step process must be followed:

- Add the text in figure 3.5 (in `ROSE/AUG0508/ROSE/projects/compass/tools/compass/emacs_compass_config.el`) to .emacs

- Change /path/to/makefile in figure 3.5 to the path to the project you are editing in Compass

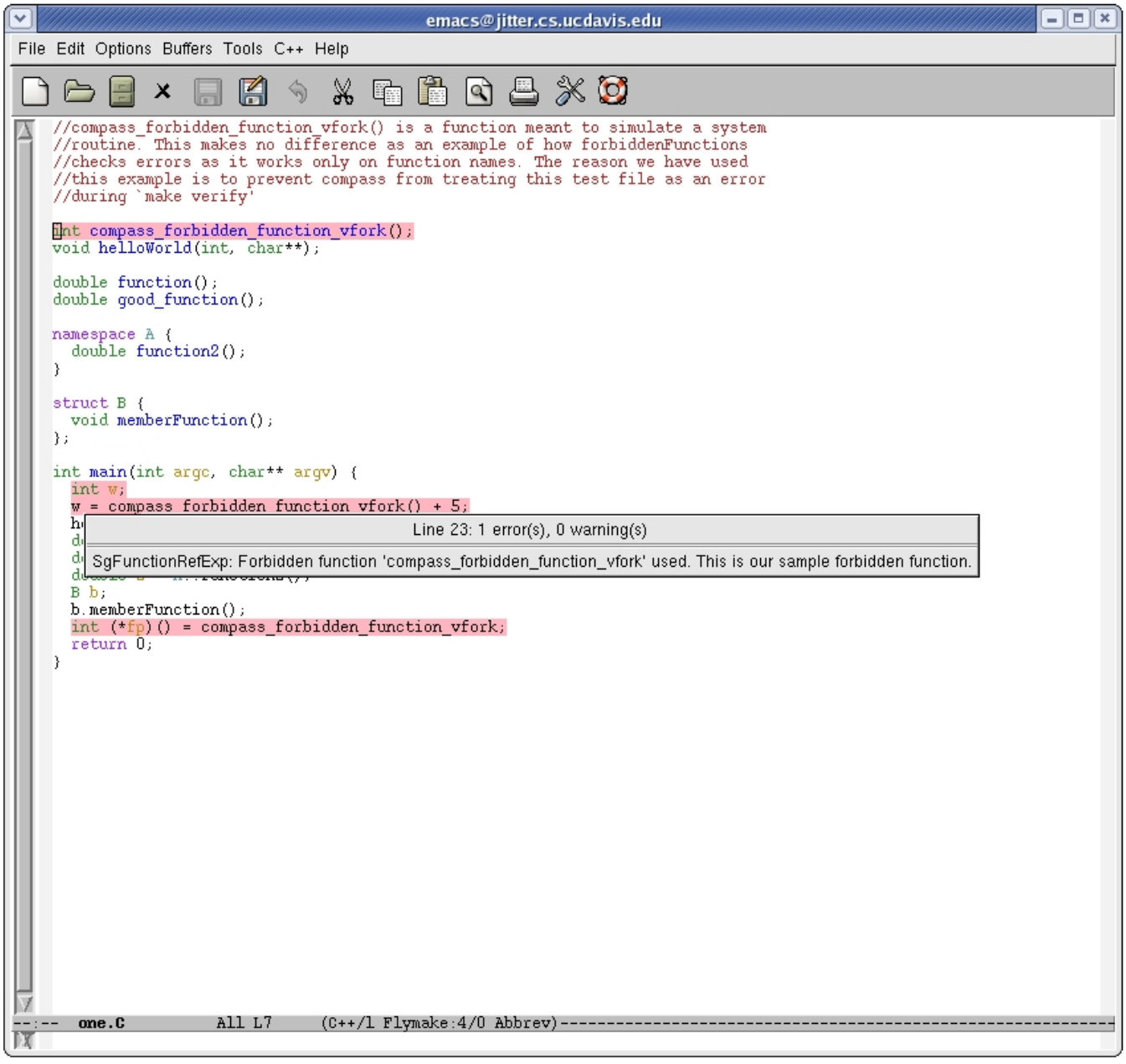
emacs@jitter.cs.ucdavis.edu
File Edit Options Buffers Tools C++ Help
//compass_forbidden_function_vfork() is a function meant to simulate a system
//routine. This makes no difference as an example of how forbiddenFunctions
//checks errors as it works only on function names. The reason we have used
//this example is to prevent compass from treating this test file as an error
//during `make verify'
int compass_forbidden_function_vfork();
void helloWorld(int, char**);
double function();
double good_function();
namespace A {
double function2();
}
struct B {
void memberFunction();
};
int main(int argc, char** argv) {
int w;
w = compass_forbidden_function_vfork() + 5;
Line 23: 1 error(s), 0 warning(s)
SgFunctionRefExp: Forbidden function 'compass_forbidden_function_vfork' used. This is our sample forbidden function.
B b;
b.memberFunction();
int (*fp)() = compass_forbidden_function_vfork;
return 0;
}
--:-- one.C All L7 (C++/l Flymake:4/0 Abbrev)

Figure 3.1: Compass error messages integrated into Emacs

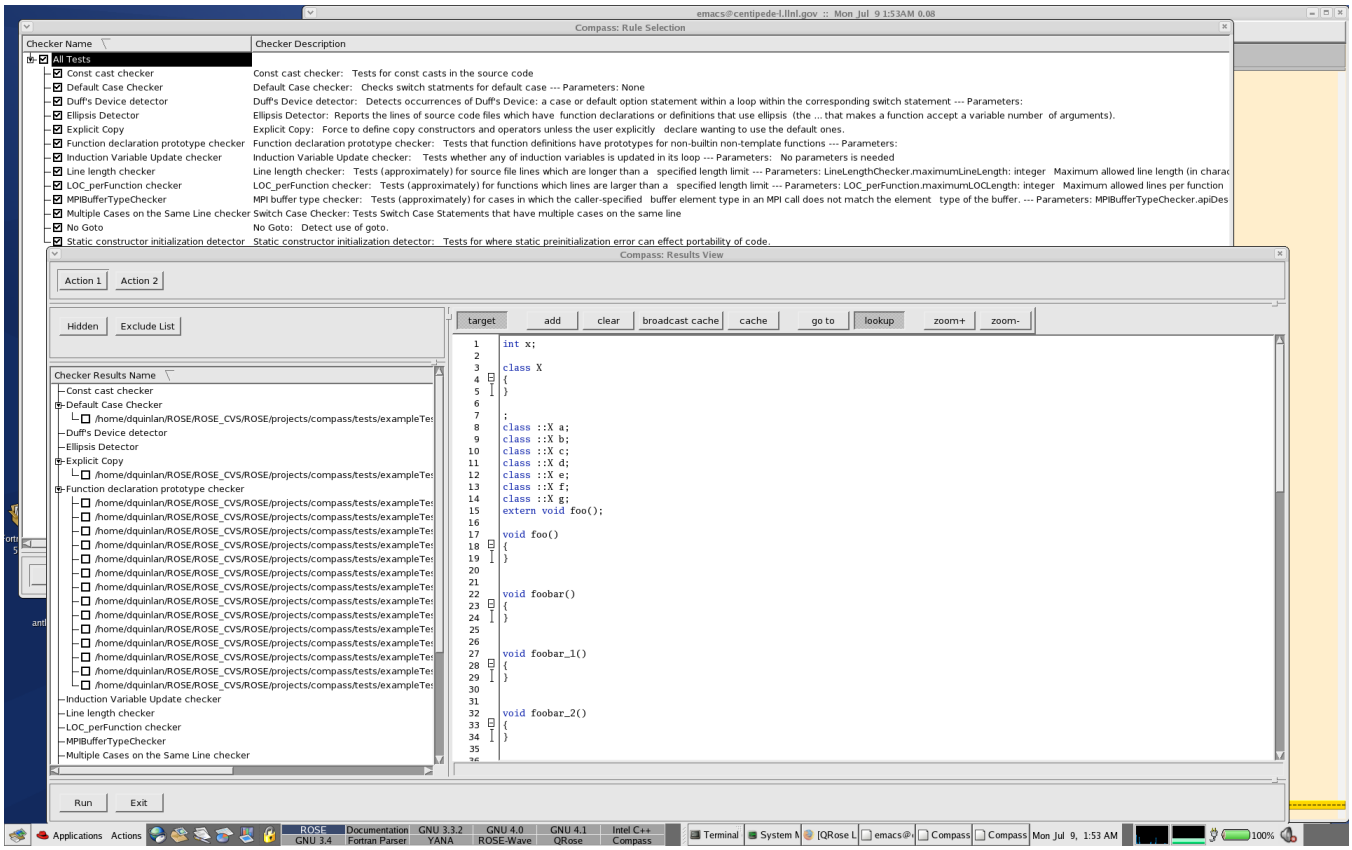


Figure 3.2: Compass GUI for interpretation of rule violations

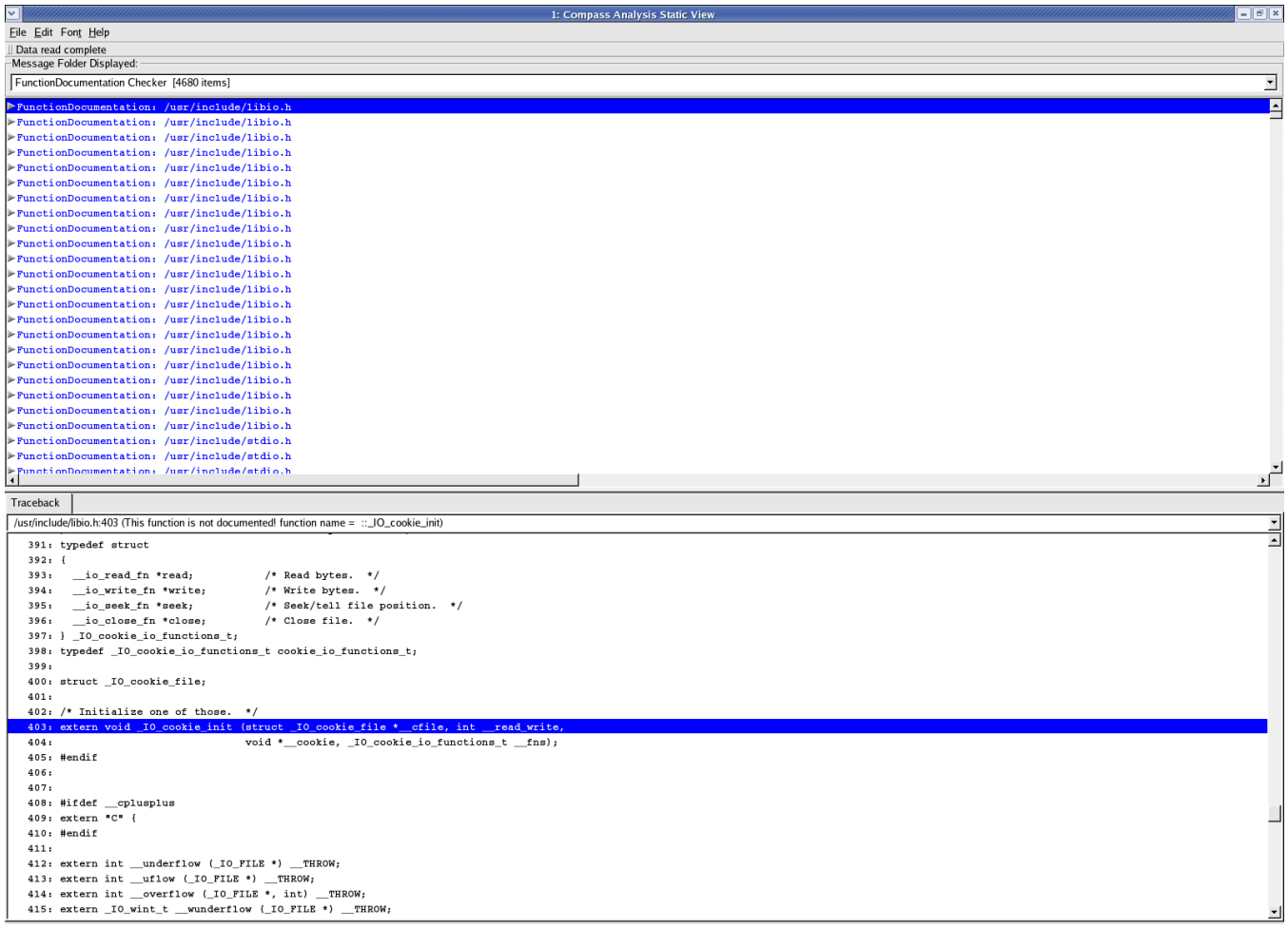


Figure 3.3: Processing of XML Compass output using ToolGear

```
LocalizedVariables: /home/ROSE/projects/compass/tests/Cxx_tests/test2006_117.C:30.5: Variable pmNull does not seem to be used.
FunctionDefinitionPrototype: /home/ROSE/projects/compass/tests/Cxx_tests/test2006_123.C:27.1-10: matching function prototype not availabl
LocPerFunction: /home/ROSE/projects/compass/tests/Cxx_tests/test2006_117.C:23.1-20: This function has too many lines of code :: LOC = 14
MagicNumber: /home/ROSE/projects/compass/tests/Cxx_tests/test2006_117.C:33.14: Occurrence of integer or floating constant.
MagicNumber: /home/ROSE/projects/compass/tests/Cxx_tests/test2006_117.C:36.14: Occurrence of integer or floating constant.
MagicNumber: /home/ROSE/projects/compass/tests/Cxx_tests/test2006_117.C:37.14: Occurrence of integer or floating constant.
FunctionDocumentation: /home/ROSE/projects/compass/tests/Cxx_tests/test2006_123.C:27.1-10: function is not documented: name =  ::main
FunctionDocumentation: /home/ROSE/projects/compass/tests/Cxx_tests/test2006_123.C:9.10: function is not documented: name =  ::X < int , i
FunctionDocumentation: /home/ROSE/projects/compass/tests/Cxx_tests/test2006_123.C:12.10: function is not documented: name =  ::X < int ,
FunctionDocumentation: /home/ROSE/projects/compass/tests/Cxx_tests/test2006_123.C:16.10: function is not documented: name =  ::X < int *
```

Figure 3.4: Example of ASCII output from Compass.

```
; New Compass support for Emacs using version 22 of Emacs and Flymake.
; Comment out these two lines to use older version of emacs.
(require 'flymake)
(setq flymake-allowed-file-name-masks (cons '(".+\\.C\\'" flymake-simple-make-init flymake-simple-cleanup flymake-get-real


(defun flymake-master-make-header-init ()
  (flymake-master-make-init 'flymake-get-include-dirs
    '("\\.C\\'" "\\.c\\'")
    "[ \t]*#[ \t]*include[ \t]*\"\\([[:word:]0-9/\\_.]*%s\\)\""))

(add-hook 'find-file-hook 'flymake-find-file-hook)

(setq flymake-log-level 3)
(setq flymake-no-changes-timeout 0.5)

(defcustom rose-source-tree "/home/dquinlan/ROSE/NEW_ROSE/" "Location of top of ROSE source tree")
(defcustom rose-build-tree "/home/dquinlan/ROSE/ROSE_CompileTree/LINUX-64bit-3.4.6/" "Location of top of ROSE build tree")
(defun add-buildfile-dir-for-rose ()
  (let ((source-dir-name (file-name-directory buffer-file-name)))
    ;(message "%S" `(source dir ,source-dir-name))
    (if
      ; (string-equal rose-source-tree (substring source-dir-name 0 (length rose-source-tree)))
        (string-equal rose-source-tree (substring source-dir-name 0 (min (length source-dir-name) (length rose-source-tree
        (let ((buildfile-dir (concat "../../../../../../../../../../../../../../../../../../../../" rose-build-tree "/" (subs
        ; (message "%S" `(buildfile-dir ,buildfile-dir))
        ; (set-variable 'flymake-buildfile-dirs (cons buildfile-dir flymake-buildfile-dirs) 'local))
          (set-variable 'flymake-buildfile-dirs (append (mapcar (lambda (dir) (concat buildfile-dir "/" dir)) flymake-buil
      (progn
        ;(message "%S" `(bad-prefix))
        source-dir-name))))
(defun set-rose-source-dir (dir) "Set the top of the ROSE source tree to use with Flymake" (interactive "DThe top of the R
  (setq rose-source-tree dir 'local)
  (add-buildfile-dir-for-rose))
(defun set-rose-build-dir (dir) "Set the top of the ROSE build tree to use with Flymake" (interactive "DThe top of the ROS
  (setq rose-build-tree dir 'local)
  (add-buildfile-dir-for-rose))

(add-hook 'find-file-hook 'add-buildfile-dir-for-rose)

;(list "make"
;;  (list "-s" "-C" "/home/dquinlan/ROSE/NEW_ROSE/developersScratchSpace/Dan/EmacsCompass_tests/"
;    (list "-s"
;   (list "-s -C" "`pwd | sed 's@^/home/dquinlan/ROSE/NEW_ROSE/@/home/dquinlan/ROSE/ROSE_CompileTree/LINUX-64bit-3.4.6/@'`
;    (concat "CHK_SOURCES=" source)
;      "SYNTAX_CHECK_MODE=1"
;    "check-syntax"))

(global-set-key [f3] 'flymake-display-err-menu-for-current-line)
(global-set-key [f4] 'flymake-goto-next-error)
```

Figure 3.5: Addition to .emacs when integrating Compass into emacs.

```
one: inc.h one.C
        g++ -c one.C
```

Figure 3.6: Example makefile before the Compass addition

```
one: inc.h one.C
        g++ -c one.C

check-syntax: inc.h one.C
        /path/to/compass/executable/compassMain -c one.C
```

Figure 3.7: Example makefile after addition to support integration of compass

- Add a 'check-syntax' rule to the makefile of the project that you are working on in Compass. This rule should compile all the files you want Compass to check or all files that you are editing with Compass as the compiler.

Figure 3.5 shows the needed changes in .emacs for integrating Compass. The last two lines are the most interesting lines since they introduce two shortkeys. [f3] can be clicked in order to display all errors for the current line while [f4] will move the cursor to the next error.

A short explanation of the code in figure 3.5 is that the first line will require the flymake extension to be available upon loading emacs while the second line will load the find-file-hook and flymake-find-file-hook functions. The "setq" sections that follows runs Compass for all files that are being edited that has the c and C extensions. The "list" section tells flymake to execute the check-syntax rule in the makefile.

#### Example check-syntax rule

Figure 3.6 shows an example makefile that compiles a file "one.C" using g++. If "one.C" is edited using emacs the addition of the "check-syntax" rule is needed, as shown in figure 3.7.

### 3.2.2 Using Compass With Vim

Compass can be used with Vim 7's QuickFix commands to display warning messages and highlight the source lines in question. A compass compiler plugin (compass.vim as shown in Figure 3.8) has been provided for Vim to parse the warning messages outputted by Compass.

Steps to make Compass work with Vim 7

- Save compass.vim into .vim/compiler. Create the target directory if it does not exist.
- Download errormarker.vim from http://www.vim.org/scripts/script.php?script_id=1861 and save it into .vim/plugin . Again, create the target directory first when it does not exist.

- Change your Makefile to use an installed compassMain as the compiler to compile your code.
- Use quickfix features of Vim 7 as documented at http://vimdoc.sourceforge.net/htmldoc/quickfix.html. Some frequently used commands are:
  - Specifying the compass plugin to use by typing a command :compiler compass
  - Open your source code using gvim and set the compiler to Compass by
  - Compile you code using compassMain, type :make
  - Display current message, type :cc
  - Display all messages, type :clist
  - Jump to next message, type :cnext

```
" Vim compiler file
" Compiler:     ROSE Compass 0.9.2a
" Maintainer:   Chunhua Liao <liao6@llnl.gov>
" Last Change:  2008 Apr. 3
"
if exists("current_compiler")
  finish
endif
let current_compiler = "compass"

if exists(":CompilerSet") != 2          " older Vim always used :setlocal
  command -nargs=* CompilerSet setlocal <args>
endif

" single line warning
" multiple line warning, %W, %C continue line %Z end of multiple line
CompilerSet errorformat=%s:\ %f:%l%.%c:\ %m,
         \%s:\ %f:%l%.%c-%*\\d:\ %m,
         \%s:\ %f:%l%.%c-%*\\d%.%*\\d:\ %m,
         \\\"%f\"\\,\ line\ %l%*\\D%c%*[^\ ]\ %m
"some notes about the error/warning message format
"Official guide: http://vimdoc.sourceforge.net/htmldoc/quickfix.html#error-file-format
" Each new rule start with a leading \ unless it is the first rule in the
" first line
"  %f: filename %l: line number %c: column number, only one is permitted %m
" actual error/warning message \ :matching a space
"  %*\\d: matching any number
```

Figure 3.8: A compiler plugin for Vim 7 compass.vim

Figure 3.9 shows an example of Compass error messages integrated into Vim 7.

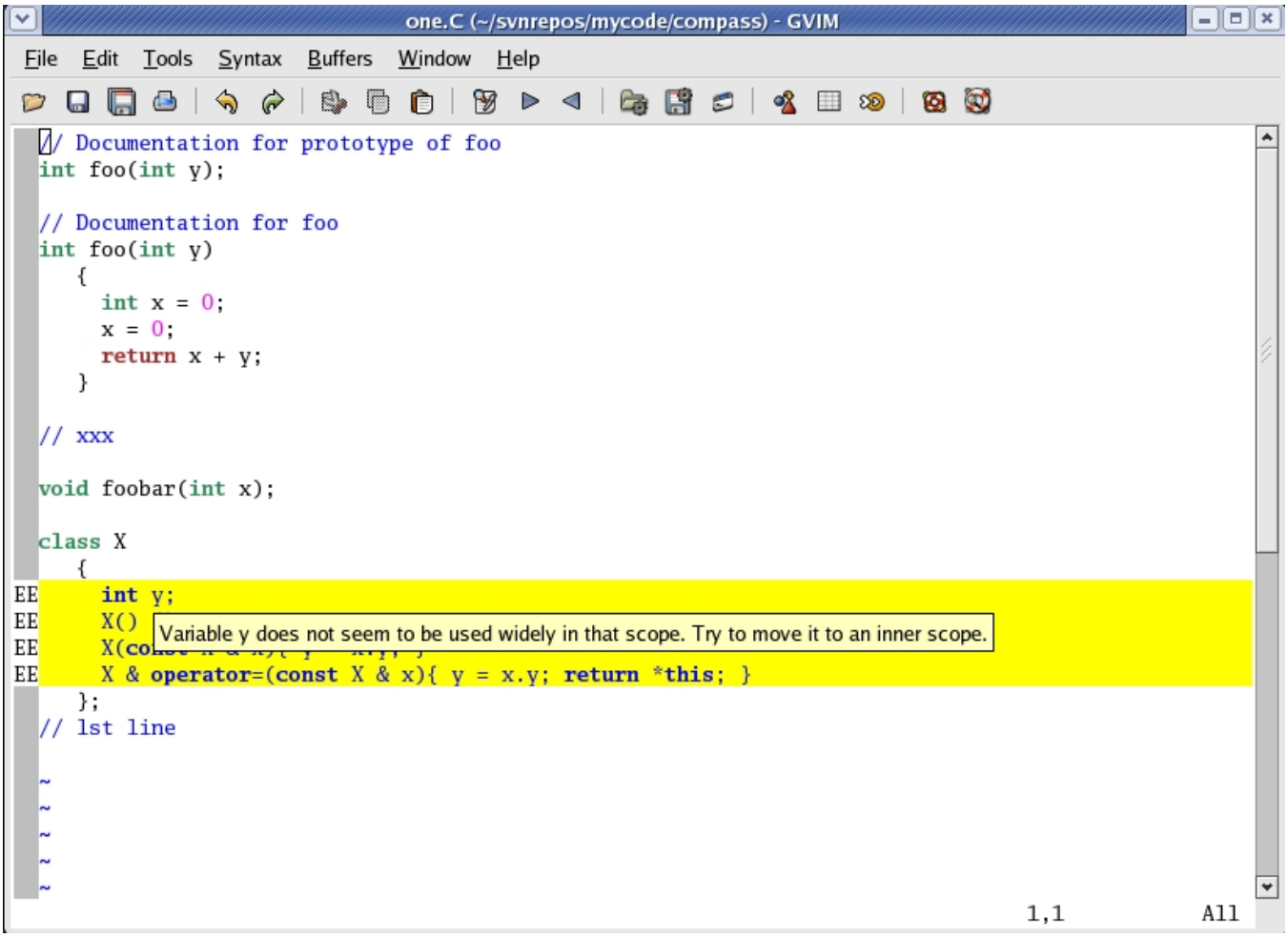


Figure 3.9: Compass error messages integrated into Vim 7

## 3.3 How To Write A New Checker

### 3.3.1 Creating A Skeleton

Compass has scripts for creating a skeleton for a new Compass checker. This skeleton can be easily adapted to write all checkers.

Follow these steps to generate a checker skeleton:

1. Enter a directory where you want the directory of your checker to be created
2. Execute `ROSE_SRC_DIR/projects/compass/src/compass_scripts/gen_checker.sh <name of your checker>`

The results of executing gen_checker.sh script is that a new directory name "multipleCasesOnSameLine" (name of your checker in camel case) is created with the following files:

```
compass.C            multipleCasesOnSameLine.C
compass.h            multipleCasesOnSameLineDocs.tex
compass_parameters   multipleCasesOnSameLine.h
compassTestMain.C    multipleCasesOnSameLine.inc
Makefile             multipleCasesOnSameLineMain.C
Makefile.am          multipleCasesOnSameLineTest1.C
```

Some of these files (compass.[Ch], compass_parameters, and compassTestMain.C) are copied from the compass_template_generator directory; while others are generated (multiple*, Makefile, Makefile.am)

It is suggested that you keep the following in mind when using gen_checker.sh:

- It is advised that you do not invoke the script gen_checker with words like checker, detector, tester, etc. Adding these verbs at the command line means that these words are added as suffixes into the directory-name. Which will make it redundant, as the compass project is about writing style-checkers!
- Some of the files have read-only permissions and are intended only for such use. Please do not change the permissions of these files.

### 3.3.2 Integrating New Checkers Into Compass Tool

The process for integrating a new checker into Compass has been automated. These directions are meant for checkers generated using gen_checker.sh. The process is similar for all compass-like tools that are built using the common instrastructure.

The steps to integrate a new checker is

1. Add <name of your checker> to CHECKER_LIST
2. Enter ROSE_BUILD_DIR/projects/compass/tools/compass
3. Execute 'make regenerate'

4. After running 'make regenerate' in the build tree then you may run make as usual.

5. Examine the RULE_SELECTION file and confirm it reflects your most recent additional checker(s) choice of execution at run-time; the default setting is "on". Please refer to section 3.5.

The file 'CHECKER_LIST' can use the '#' comment delimiter at the beginning of any checker name to remove that checker from compilation. The hash mark may only appear at the beginning of the line. The compass_submission_setup.sh script must be run again with the "regenerate" option if any checkers are commented out. **Note that no space is permitted between the '#' and the name of the checker.**

Additionally, a blank checker may be automatically integrated into the current compass tool as a method to test for build system errors. To execute this test ,type:

```
make testNewChecker
```

from any compass tool directory. This test will create a blank checker from the template attempt to install the checker then uninstall it thus leaving the tool in its previous state.

## 3.4 Including/Excluding Checkers in the Compass Build Process

This section describes how to select which of the checkers to integrate into Compass out of all the checkers available in source form in the Compass source directory. For security reasons Compass uses this static build process since it is a central goal of Compass that it should run as a trusted part of a project's build process. If the integration of checkers had been automatic through a dynamic plugin mechanism it would be hard to ensure that the dynamic list of checkers was secure, but for a static list of trusted checkers this is possible.

In order to integrate a checker into Compass the user must:

- Add the name of the checkers directory in the CHECKER_LIST in the compass source directory

- Run `make regenerate` in the Compass build directory

If a user or developer intends to integrate a new checker into the Compass source tree refer to section 3.3.2.

Usually the CHECKER_LIST is only modified when a user or developer wants to add a new checker or select a subset of trusted checkers. Checkers can be commented out using a `#`, no space is allowed between the `#` and the checker name.

## 3.5 Including/Excluding Checkers During Compass Execution

This section describes how to execute a subset of the checkers provided in the build process (see section 3.4). This process is significantly more interactive than defining what checkers to include in the Compass build process. Since it is not unheard of that rules implemented by different checkers can be mutually exclusive or even contradicting this mechanism is essential for selecting the subset of checkers that are interesting for a specific program that is checked. Separate projects of developers could easily have their own RULE_SELECTION file to permit high levels of customization in the use of a Compass tool containing a large number of checkers (e.g. for different languages).

When used with the Emacs interface this provides a simple way to turn on and off specific checkers by editing a single file (RULE_SELECTION). The name of this file is specified in the `compass_parameters` file, this name may be changed. The directories searched are: current directory, user home directory, and Compass source tree (respectively).

```
Compass.RuleSelection=RULE_SELECTION
```

In order to select a checker to run the user must:

- Add a line '-:<name of checker>' in a file called RULE_SELECTION.
- If a line '-:<name of checker>' already exist the '-' can be modified into a '+' to enable the checker or into a '-' to disable a checker.

It is required that every checker integrated into the Compass build is mentioned in the RULE_SELECTION.

## 3.6 Including/Excluding Paths and Filenames with Compass

Compass permits paths and filenames to be specified for inclusion/exclusion in reporting checker rule violations. Run `compassMain --help` to see the commandline options. Numerous other commandline options provided for all tools build using ROSE may also be relevant.

## 3.7 Checking Security Properties of Checkers

Compass is designed for extensibility while providing the security for the codes being checked. To support this Compass provides a simple mechanism for verifying specific properties of the checkers used in Compass. Compass implements a specific small number of checkers that are used for checking the checkers in Compass. The directory compassVerify contains the implementation of this subset of Compass that is used on itself. These checkers may not be modified and in the future MD-5 checksums will be provided to ensure the integrity of this subset of Compass. To verify the compass checkers run:

- `make verify`
  This makefile rule runs the compassVerify/compassMain on all the source files in all the checkers directories in Compass. Because it runs compass on so many separate files this step can take a long time.

- Or `make oneBigVerify`
  The makefile rule runs the compassVerify/compassMain on a single generated file built from all the checker source files and is particularly quick to run.

## 3.8 Testing Compass and its Checkers

The `tests` directory contains directories of tests that are language specific:

- C_tests
  This directory contains a Makefile which will use the ROSE C test codes to test Compass.

- Cxx_tests
  This directory contains a Makefile which will use the ROSE C++ test codes to test Compass.

To run these tests type `make check` at any level on the Compass directory hierarchy of the build tree.

## 3.9 Extending the Compass Infrastructure

Compass, as well as being a tool for software analysis, is also a capable infrastructure for building other tools like Compass that utilizes the work put into Compass checkers. There are many reasons why one would like to have a separate executable tool beneath Compass rather than simply customizing a particular build of the Compass tool. An individual or organization may consider a specific subset of Compass checker rules to be particularly relevant; thus would like to have a custom tool to check only these rules. Another scenario may find that the particular interface for Compass through the command line needs to be changed; but, it is not advisable to alter Compass main directly!

This section will demonstrate how to extend the Compass infrastructure and checkers to build a separate executable tool beneath the Compass project. A simple tutorial will detail the steps necessary to instantiate the Compass infrastructure. The these steps have been followed and the mechanism is demonstrated in an example directory (`sampleCompassSubset`; this directory can be copied.

Suppose one wishes to build a version of Compass with only those checker rules authored by an individual for debugging purposes. Certainly, it would not be desirable for the main Compass tool to only consist of this subset of rules; yet this specialized version might be required regularly enough for repeatedly altering of the static or dynamic checker rule selection files to become troublesome. The solution is to use the Compass infrstructure to build

a separate tool by creating a subdirectory under Compass with a small portion of Compass infrastructure files. Only four files are required: compassMain.C, CHECKER_LIST, RULE_SELECTION, Makefile.am. Note also that the `compassMain.C` file can be any name, but must only be consistant with the `Makefile.am`. This example of using the compass infrastructure is compiled as part of compiling compass and defines a compass-infrastructure-based tool that implements about 25 randomly selected checkers (form the collection in `projects/compass/extensions/checkers`).

Assume that the present working directory is `projects/compass/tools` of the ROSE source tree. First, create a directory for the new Compass subset tool.

```
mkdir sampleCompassSubset
```

then add this directory in the `SUBDIRS` variable of the automake file `projects/compass/tools/Makefile.am`. Additionally, introduce this new Makefile into the top level source tree `configure.in` file. A snippet concerning these changes is given below:

```
projects/compass/tools/Makefile.am:
SUBDIRS = compass compassVerifier sampleCompassSubset

configure.in:
projects/compass/tools/sampleCompassSubset/Makefile
projects/compass/tools/compassVerifier/Makefile
```

*Note that after any modification of* `configure.in`*, the configure command for ROSE will have to be rerun (else the makefile will cause it to be called for you).*

To specify how this tool is to be configured and built create a `Makefile.am`, `projects/compass/tools/sampleCompassSubset/Makefile.am` is provided as an example. This example can be a template and may be used in general for any Compass-like tool. Any tool that is an extension of the Compass infrastructure will reuse much of the Compass infrastructure such as `compassSupport`. Furthermore, much of the Compass build process requires the automatic generation of files in the build tree such as `compass_makefile.inc`, `compass_parameters`, etc. Thus an include file has been provided in Compass to define these common rules– `compass.inc`; and is included in this `Makefile.am`. A more advanced implementation of this automake file template is located in Compass Verifier; which is itself another tool extended off Compass and designed to check the rules implemented in Compass.

So far, a blank Compass-like tool has been created and configured to build with ROSE. Still missing are a few Compass files that produce a successful compile and linked executable. The next step is to populate the empty Compass-like tool with checkers from Compass. Note that all checkers reside as subdirectories in `projects/compass/extensions/checkers`; but that subsets of these are selected for use, thus all checkers are always optionally available to all compass-infrastructure-based tools. Create a new file called `CHECKER_LIST` under the `sampleCompassSubset` directory (an example has been provided) and insert the names of the desired checkers into this file.

One may also create the dynamic rule selection file `RULE_SELECTION` or defer this file to be automatically generated with all checker rules activated by default. Finally, it is necessary to write or copy the `compassMain.C` file that defines (among other function) the command line processing options where one may alter the interface. For `sampleCompassSubset` the `compassMain.C` of `projects/compass/src/compassSupport` is symbolically linked in place. The complete directory list for `sampleCompassSubset` now looks like

```
CHECKER_LIST
compassMain.C -> ../../src/compassSupport/compassMain.C
Makefile.am
RULE_SELECTION
```

The source tree creation and configuration of a new Compass-like tool is complete with `Makefile.am`, `CHECKER_LIST`, `RULE_SELECTION`, and `compassMain.C` (rename `compassMain.C` as you like just make sure that the name is consistant with the `Makefile.am`). Build and configure ROSE as usual and then invoke Make at the top level build tree or `projects/compass` subdirectory to compile and link the new Compass-like tool executable. In this tutorial example the new target location is `projects/compass/tools/sampleCompassSubset`.

# Chapter 4

# Using Compass Verifier

Compass Verifier is a tool extended from Compass to analyze the source code of Compass and its checkers to detect certain properties. The motivation and design of this tool are discussed in Chapter 2 (Design and Verification). This chapter will detail the Make rules written to run Compass Verifier and to setup the parameters to the AllowedFunctions checker.

The rules to setup and run Compass Verifier are found in `projects/compass/tools/compassVerifier/Makefile.am`. Follows is an overview of their usage labels, (`make oneBigVerify, verify, ...`). Verifier is designed to examine the compass tool referenced in the environment variable `$(TOOLBUILD)` which by default is `projects/compass/tools/compass`. This variable may be changed in the `Makefile.am` of the source directory or on the command line to `make` in the build tree.

- `oneBigVerify:` a fast but rough static analysis over the Compass checker sources. This rule executes `compassVerifier` over the concatenation of all sources with names matching those checkers listed in `$(TOOLBUILD)/CHECKER_LIST_WITHOUT_COMMENTS` with extension (`.C`). These should be the only sources from the checker subdirectories that are built with Compass. The consistency of this concatenation depends on no global (function, variable, etc.) declaration conflicts; usually, this is perserved by default as all checkers employ their own namespace.

- `verify:` a slow and more complete analysis over all files found in the checker directories listed in `$(TOOLBUILD)/CHECKER_LIST_WITHOUT_COMMENTS`. The individual rule labels for verify are generated in `verify.makefile` identically to those lower case names found in `$(TOOLBUILD)/CHECKER_LIST_WITHOUT_COMMENTS`. Each rule logs the standard output and error to files suffixed `_out.txt` and `_err.txt` respectively and the measure of a passing checker is an empty standard error log file. Concurrency may be used with (`-j#`) option to GNU Make to reduce the time to complete verify.

- `new_allow_list:` regenerates the parameter file `projects/compass/extensions/checkers/allowedFunctions/compass_parameters` in the source tree used as the allowed functions list in Compass Verifier.

The present list assumes that all sources found in the present source tree of Compass are trusted in their use of function calls and whatever functions are referenced are allowed. Sources processed by Compass Verifier to build the list of allowed functions consist of compass.C, compassTestMain.C, buildCheckers.C, and the concatenation of checker sources identical to that of `oneBigVerify`. Please see the documentation for allowedFunctions 5.2 for the details of how the new list of allowed functions is created and maintained. *Not implemented: we plan to limit the recursion of function references to exclude those functions called by calls to library functions. This should greatly reduce the number of allowed functions such that an individual may inspect the generated file.*

These three Make rules describe the basic user interface for Compass Verifier. Other parameters such as those for ForbiddenFunctions should be changed manually. And an individual should examine the results of Compass Verifier to ensure a checker meets the validation standards sought-after.

# Chapter 5

# Compass Checkers

## 5.1 Allocate And Free Memory In The Same Module At The Same Level Of Abstraction

Allocating and freeing memory in different modules and levels of abstraction burdens the programmer with tracking the lifetime of that block of memory. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to programming defects such as double-free vulnerabilities, accessing freed memory, or writing to unallocated memory.

To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

The affects of not following this recommendation are best demonstrated by an actual vulnerability. Freeing memory in different modules resulted in a vulnerability in MIT Kerberos 5 MITKRB5-SA-2004-002 . The problem was that the MIT Kerberos 5 code contained error-handling logic, which freed memory allocated by the ASN.1 decoders if pointers to the allocated memory were non-NULL. However, if a detectable error occured, the ASN.1 decoders freed the memory that they had allocated. When some library functions received errors from the ASN.1 decoders, they also attempted to free, causing a double-free vulnerability.

### 5.1.1 Parameter Requirements

No Parameter specifications.

### 5.1.2 Implementation

No implementation yet!

### 5.1.3 Non-Compliant Code Example

This example demonstrates an error that can occur when memory is freed in different functions. The array, which is referred to by list and its size, number, are then passed to the verify_list() function. If the number of elements in the array is less than the value MIN_SIZE_ALLOWED, list is processed. Otherwise, it is assumed an error has occurred, list is freed, and the function returns. If the error occurs in verify_list(), the dynamic memory referred to by list will be freed twice: once in verify_list() and again at the end of process_list().

```
% write your non-compliant code example
int verify_size(char *list, size_t list_size) {
  if (size < MIN_SIZE_ALLOWED) {
    /* Handle Error Condition */
    free(list);
    return -1;
  }
  return 0;
}
```

```
void process_list(size_t number) {
  char *list = malloc(number);

  if (list == NULL) {
    /* Handle Allocation Error */
  }

  if (verify_size(list, number) == -1) {
    /* Handle Error */

  }

  /* Continue Processing list */

  free(list);
}
```

### 5.1.4 Compliant Solution

To correct this problem, the logic in the error handling code in verify_list() should be changed so that it no longer frees list. This change ensures that list is freed only once, in process_list().

```
% write your compliant code example
int verify_size(char *list, size_t list_size) {
  if (size < MIN_SIZE_ALLOWED) {
    /* Handle Error Condition */
    return -1;
  }
  return 0;
}

void process_list(size_t number) {
  char *list = malloc(number);

  if (list == NULL) {
    /* Handle Allocation Error */
  }

  if (verify_size(list, number) == -1) {
    /* Handle Error */
  }

  /* Continue Processing list */

  free(list);
}
```

### 5.1.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Write your checker algorithm

### 5.1.6 References

ISO/IEC9899-1999 MEM00-A. Allocate and free memory in the same module, at the same level of abstraction

## 5.2 Allowed Functions

The validation of code properties often requires the detection of certain functions used. In some cases, the list of undesirable functions is too large for a mechanism like forbidden functions to be effective. This allowed functions checker performs the opposite by filtering function references against a premade accepted list. Only those function on this list are allowed in the source and any other function reference used is a reported violation. Usually, the list of allowed functions is generated based off "trusted sources" input to the checker.

### 5.2.1 Parameter Requirements

The compass_parameter requirement for this checker has several options including

- AllowedFunctions.OutFile specifies the file name path for the output of this checker. The list of all allowed functions are written to this file formatted identically to compass_parameters. All other options passed to allowedFunctions by compass_parameters is also preserved in this output file.

- AllowedFunctions.Function# appends an allowed function given in the manged name format written by this checker. The parameters of this format must start sequentially at zero and increment until FunctionNum minus one.

- AllowedFunctions.FunctionNum a simultaneous boolean flag and state variable; specifies the total number of allowed functions in the accepted list. If this number is postive, the allowed function checker executes in testing mode, where all new function references are written to OutFile as new allowed functions. A negative value for FunctionNum switches the checker into detection mode where all function references not read from the list in compass_parameters are flagged as violations.

- AllowedFunctions.Library# adds to a list of paths treated as safe libraries. Any sources found in these paths will be ignored by allowedFunctions.

### 5.2.2 Implementation

This checker functions in two modes depending on the boolean sign of the AllowedFunctions.FunctionNum parameter.

Positive signed FunctionNum puts this checker into testing mode where new function references are added to the list of allowed functions. This mode begins by reading the compass_parameters file for found allowed functions and other options given to allowedFunctions. The existing list of allowed functions is output in-order to the output file specified by OutFile. The allowedFunctions traversal then begins by visiting all function and member function references and detecting their membership in the set of allowed functions. Functions not found in this set are added and written to the OutFile while those already found do nothing.

The function reference is detected and output using a special mangled string generated by allowedFunctions that serves as its unique identifier. This format is recursively generated by looking at the types of a functions return and argument parameters as well as its qualified name joined in a comma-separated string. The string sequence is return type, qualified name with scope, and arguments or void, etc. One simple example is for the `memchr` function

```
AllowedFunctions.Function39=*void,::memchr,*void,int,size_t,
```

After all existing and new allowed functions are written, the FunctionNum parameter is updated with the current number of allowed functions in the list given in OutFile. Essentially, the testing mode is designed for multiple executions of compass with different sources such that a list of allowed functions grows and is maintained in a format usable for compass_parameters.

The other operation mode (detection), reads from compass_parameters the set of allowed functions. Similarly, the traversal visits all function and member function references and constructs the unique manged string identifier for the function. This mangled string is looked up in the set of all allowed functions. If the function is found then execution continues; but a function that is not found in the set of allowed functions is reported as a violation. In general, this will be the mode most users will run AllowedFunctions under.

Several exclusion mechanism are implemented by allowedFunctions for ignoring local function definitions and function calls occuring from source designated as a safe library. Their implementation relies on the file classification mechanism in ROSE string support to distinguish between user, system, and library sources.

## 5.3 Assignment Operator Check Self

This test checks to make sure that the first statement in a assignment operator is a check for self-assignment. As noted in An Abbreviated C++ Code Inspection Checklist 12.1.3 . This will save time allocating new memory and (hopefully) deleting the previous copy. The check for return this; is handled by another checker.

### 5.3.1 Parameter Requirements

No parameters required.

### 5.3.2 Implementation

This test checks to make sure that the first statement in a assignment operator is a check for self-assignment. As noted in An Abbreviated C++ Code Inspection Checklist 12.1.3 . This will save time allocating new memory and (hopefully) deleting the previous copy. The check for return this; is handled by another checker.

### 5.3.3 Non-Compliant Code Example

```
class bike
{
public:
 const bike& operator= (const bike& other);
};

const bike& bike::operator= (const bike& other)
{
...
return *this;
}
```

### 5.3.4 Compliant Solution

```
const bike& bike::operator= (const bike& other)
{
if (this == &other)
    {return *this;
    }
...
return *this;
}
```

### 5.3.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Identify member function
2. Check name for operator=
3. Check first statement as If Statement
4. Check arguments to expression to be this and the right hand argument.

### 5.3.6 References

Abbreviated Code Inspection Checklist Section 12.1.3, Assignment Operator"

## 5.4 Assignment Return Const This

Here we check to make sure that all assignment operators (operator=) return const classType&. By making the return a reference we can use a = b = c; which is legal C++. By making the reference const we prevent (a = b) = c; which is illegal C++.

### 5.4.1 Parameter Requirements

No parameters necessary.

### 5.4.2 Implementation

Every member function is checked to see if the name matches 'operator='. If so we check to make sure the return type is const nameofclass&. All three (const, ref, classname) must be present. We then make sure there is at least one explicit return of *this and no explicit returns of anything else. Note: At this time we do not make sure that all paths must reach an explicit return. This is, however, already a warning in ROSE when there is not an explicit return for a non-void returning function. There is also another checker to ensure explicit returns.

### 5.4.3 Non-Compliant Code Example

```
class smallCat
{
smallCat& operator=(smallCat& other);
}

smallCat& smallCat::operator=(smallCat& other)
{
...
}
```

### 5.4.4 Compliant Solution

```
class smallCat
{
const smallCat& operator=(smallCat& other);
}

const smallCat& smallCat::operator=(smallCat& other)
{
...
}
```

### 5.4.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Identify member function
2. Check Name for operator=
3. Get return type
4. check for typename and const
5. find explicit return and check for this.

### 5.4.6 References

Abbreviated Code Inspection Checklist Section 12.1.4, Assignment Operator"

## 5.5 Avoid Using The Same Handler For Multiple Signals

It is possible to safely use the same handler for multiple signals, but doing so increases the likelihood of a security vulnerability. The delivered signal is masked and is not delivered until the registered signal handler exits. However, if this same handler is registered to handle a different signal, execution of the handler may be interrupted by this new signal. If a signal handler is constructed with the expectation that it cannot be interrupted, a vulnerability might exist. To eliminate this attack vector, each signal handler should be registered to handle only one type of signal.

### 5.5.1 Parameter Requirements

No Parameter specifications.

### 5.5.2 Implementation

No implementation yet!

### 5.5.3 Non-Compliant Code Example

This non-compliant program registers a single signal handler to process both SIGUSR1 and SIGUSR2. The variable sig2 should be set to one if one or more SIGUSR1 signals are followed by SIGUSR2.

```
% write your non-compliant code example
#include <signal.h>
#include <stdlib.h>
#include <string.h>

volatile sig_atomic_t sig1 = 0;
volatile sig_atomic_t sig2 = 0;

void handler(int signum) {
  if (sig1) {
     sig2 = 1;
  }
  if (signum == SIGUSR1) {
    sig1 = 1;
  }
}

int main(void) {
  signal(SIGUSR1, handler);
  signal(SIGUSR2, handler);

  while (1) {
```

```
    if (sig2) break;
    sleep(SLEEP_TIME);
  }

  /* ... */

  return 0;
}
```

The problem with this code is that there is a race condition in the implementation of handler(). If handler() is called to handle SIGUSR1 and is interrupted to handle SIGUSR2, it is possible that sig2 will not be set. This non-compliant code example also violates SIG31-C. Do not access or modify shared objects in signal handlers.

### 5.5.4 Compliant Solution

This compliant solution registers two separate signal handlers to process SIGUSR1 and SIGUSR2. The sig1_handler() handler waits for SIGUSER1. After this signal occurs, the sig2_handler() is registered to handle SIGUSER2. This solution is fully compliant and accomplishes the goal of detecting whether one or more SIGUSR1 signals are followed by SIGUSR2.

```
% write your compliant code example
#include <signal.h>
#include <stdlib.h>
#include <string.h>

volatile sig_atomic_t sig1 = 0;
volatile sig_atomic_t sig2 = 0;

void sig1_handler(int signum) {
  sig1 = 1;
}

void sig2_handler(int signum) {
  sig2 = 1;
}

int main(void) {
  signal(SIGUSR1, handler);

  while (1) {
    if (sig1) break;
    sleep(SLEEP_TIME);
  }
```

```
  signal(SIGUSR2, handler);
  while (1) {
    if (sig2) break;
    sleep(SLEEP_TIME);
  }

  /* ... */

  return 0;
}
```

### 5.5.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Write your checker algorithm

### 5.5.6 References

ISO/IEC 03 SIG00-A. Avoid using the same handler for multiple signals

## 5.6 Binary Buffer Overflow

### 5.6.1 Parameter Requirements

No Parameter specifications yet!

### 5.6.2 Implementation

No implementation yet!

### 5.6.3 Non-Compliant Code Example

```
% write your non-compliant code example
```

### 5.6.4 Compliant Solution

```
% write your compliant code example
```

### 5.6.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Write your checker algorithm

### 5.6.6 References

## 5.7 Binary Interrupt Analysis

### 5.7.1 Parameter Requirements

No Parameter specifications yet!

### 5.7.2 Implementation

No implementation yet!

### 5.7.3 Non-Compliant Code Example

```
% write your non-compliant code example
```

### 5.7.4 Compliant Solution

```
% write your compliant code example
```

### 5.7.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Write your checker algorithm

### 5.7.6 References

## 5.8 Bin Print Asm Functions

### 5.8.1 Parameter Requirements

No Parameter specifications yet!

### 5.8.2 Implementation

No implementation yet!

### 5.8.3 Non-Compliant Code Example

```
% write your non-compliant code example
```

### 5.8.4 Compliant Solution

```
% write your compliant code example
```

### 5.8.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Write your checker algorithm

### 5.8.6 References

## 5.9 Bin Print Asm Instruction

### 5.9.1 Parameter Requirements

No Parameter specifications yet!

### 5.9.2 Implementation

No implementation yet!

### 5.9.3 Non-Compliant Code Example

```
% write your non-compliant code example
```

### 5.9.4 Compliant Solution

```
% write your compliant code example
```

### 5.9.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Write your checker algorithm

### 5.9.6 References

## 5.10 Boolean Is Has

This checker makes sure that all boolean variables and functions that return a boolean are all named following the convention 'is_" or 'has_" per the ALE3D manual.

### 5.10.1 Parameter Requirements

No parameters required.

### 5.10.2 Implementation

This implementation checks to see if a function returns a boolean, if so it checks the first 4 characters in the name of the function for 'is_" or 'has_". It also checks all variable declarations, checks for boolean type, then does the same substring match on its name.

### 5.10.3 Non-Compliant Code Example

```
bool chosen_poorly ()
{
return true;
}

int main()
{
bool badly_named;
return 0;
}
```

### 5.10.4 Compliant Solution

```
bool has_chosen_poorly ()
{
return true;
}

int main()
{
bool is_badly_named;
return 0;
}
```

### 5.10.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. finds bool declarations or bool return function declarations
2. checks name

### 5.10.6 References

ALE3D Section 3.6 Booleans”

## 5.11 [No Reference] Buffer Overflow Functions

This analysis detects possible buffer overflows due to the usage of 'unsafe' function calls. The results need to be either inspected by the user or if applicable, unsafe function calls can be exchanged against their safe counterparts.

### 5.11.1 Non-Compliant Code Examples

```
1 #include <stdio.h>
2 #include <string.h>
3
4 using namespace std;
5
6 void fail() {
7   char string[50];
8   int file_number = 0;
9   sprintf( string, "file.%d", file_number );
10
11  char result[100];
12  float fnum = 3.14159;
13  sprintf( result, "%f", fnum );
14
15
16  char str1[]="Sample string";
17  char str2[40];
18  char str3[40];
19  memcpy (str2,str1,strlen(str1)+1);
20  memcpy (str3,"copy successful",16);
21  printf ("str1: %s\nstr2: %s\nstr3: %s\n",str1,str2,str3);
22
23 }
```

### 5.11.2 Compliant Solution

Example as above; use snprintf instead of sprintf.

### 5.11.3 Parameter Requirements

None.

### 5.11.4 Implementation

The following functions are checked for

- sprintf
- scanf
- sscanf
- gets
- strcpy
- _mbscpy
- lstrcat

- memcpy
- strcat

### 5.11.5 References

Foster , "James C.Foster, Vitaly Osipov, Nish Bhalla, Niels Heinen, Buffer Overflow Attacks, ISBN 1-932266-67-4, p. 211"

## 5.12 Secure Coding : EXP04-A. Do not perform byte-by-byte comparisons between structures

Structures may be padded with data to ensure that they are properly aligned in memory. The contents of the padding, and the amount of padding added is implementation defined. This can can lead to incorrect results when attempting a byte-by-byte comparison between structures.

### 5.12.1 Non-Compliant Code Example

This example uses `memcmp()` to compare two structures. If the structures are determined to be equal, `buf_compare()` should return 1 otherwise, 0 should be returned. However, structure padding may cause `memcmp()` to evaluate the structures to be unequal regardless of the contents of their fields.

```
1 struct my_buf {
2   size_t size;
3   char buffer[50];
4 };
5
6 unsigned int buf_compare(struct my_buf *s1, struct my_buf *s2) {
7   if (!memcmp(s1, s2, sizeof(struct my_struct))) {
8     return 1;
9   }
10  return 0;
11 }
12
```

### 5.12.2 Compliant Solution

To accurately compare structures it is necessary to perform a field-by-field comparison [ Summit 95 ]. The `buf_compare()` function has been rewritten to do this.

```
1 struct my_buf {
2   size_t size;
3   char buffer[50];
4 };
5
6 unsigned int buf_compare(struct buffer *s1, struct buffer *s2) {
7   if (s1->size != s2->size) return 0;
8   if (strcmp(s1->buffer, s2->buffer) != 0) return 0;
9   return 1;
10 }
11
```

### 5.12.3 Risk Assessment

Failure to correctly compare structure can lead to unexpected program behavior.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---|---|---|---|---|---|
| EXP04-A | **2** (medium) | **1** (unlikely) | **1** (high) | **P2** | **L3** |

#### Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website .

### 5.12.4 References

[ Dowd 06 ] Chapter 6, "C Language Issues" (Structure Padding 284-287) [ ISO/IEC 9899-1999 ] Section 6.7.2.1, "Structure and union specifiers" [ Kerrighan 88 ] Chapter 6, "Structures" (Structures and Functions 129) [ Summit 95 ] comp.lang.c FAQ list - Question 2.8

## 5.13 Char Star For String

This checker will report when STL strings are used.

### 5.13.1 Parameter Requirements

The checker does not take any parameters.

### 5.13.2 Implementation

The checker finds variables declarations, function arguments and typedefs of string type. The string type may be of pointer type, reference type, array type or it can be modified without any problems.

### 5.13.3 Non-Compliant Code Example

```
#include <string>

typedef std::string string2;
void bar(std::string arg1){

};

int main(){
   std::string  foo1;
   std::string* foo2;
   std::string  foo4[4];

};
```

### 5.13.4 Compliant Solution

```
typedef char* string2;
void bar(char* arg1){

};

int main(){
   char*  foo1;
   char** foo2;
   char   foo4[4];
};
```

### 5.13.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Traverse the AST
2. If a variable declaration, functions argument or typedef has a string base type report an error.

### 5.13.6 References

The ALE3D style guide section 16.2 states that C strings must be used instead of STL strings due to portability problems.

## 5.14 Comma Operator

The comma operator is commonly considered confusing without any redeeming value. This checker makes sure that it is not used. It reports any use of the built-in comma operator and any declaration of an overloaded comma operator.

### 5.14.1 Parameter Requirements

This checker does not require any parameters.

### 5.14.2 Non-Compliant Code Example

```
int f_noncompliant(int n)
{
    return (n++, n++, n); // not OK (twice): comma operator
}
```

### 5.14.3 Compliant Solution

```
int f_compliant(int n)
{
    n++;
    n++;
    return n;
}
```

### 5.14.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers identifying any appearance of the built-in comma operator and any declaration of a function overloading the comma operator.

### 5.14.5 References

A reference to this pattern is: The Programming Research Group: "High-Integrity C++ Coding Standard Manual", Item 10.19: "Do not use the comma operator."

## 5.15 [No Reference] Computational Functions

This analysis computes the amount of floating point, integer, floating point pointer and integer pointer operations within each function. If the value is larger than specified, than a warning is triggered. The analysis helps to identify functions with high computatuional value.

### 5.15.1 Non-Compliant Code Examples

```
void fail() {
  int x=4;
  int y=x+5+7;
  int *z = &x;
  y = *z+*z+6+8+9;
}
```

### 5.15.2 Compliant Solution

```
void pass() {
  int x= 4;
  int y = x+5+7;
}
```

### 5.15.3 Parameter Requirements

computationalFunctions.maxIntOps defines the maximum of integer operations permitted. computationalFunctions.maxFloatOps defines the maximum of floating point operations permitted.

### 5.15.4 Implementation

The implementation checks for the following direct types:

- SgAddOp
- SgSubtractOp
- SgDivideOp
- SgMultiplyOp

The implementation checks for the following indirect types:

- SgCastExp - operations hidden behind cast
- SgVarRefExp - variable operations
- SgPointerDerefExp - pointer operations
- SgPntrArrRefExp - array operations

### 5.15.5 References

## 5.16 Const Cast

Casting the constness away should never be done.

Casting away constness via const_cast is just plain false advertising. If a member function's signature is void someFunc(const foo& arg); then the function advertises to it's clients that it will not call any non-const member functions on the arg. Casting the constness of arg away to allow the use of non-const member functions can create unexpected results for clients of this function.

### 5.16.1 Parameter Requirements

No paramaters are needed.

### 5.16.2 Implementation

The checker inspects every cast in the AST. If the type casted to is equal to the type casted from minus the const-modifier it is an error.

### 5.16.3 Non-Compliant Code Example

```
void foo(){
   const int x = 2;
   int y = (int) x;
}
```

### 5.16.4 Compliant Solution

```
void foo(){
   int x = 2;
   int y = x;
}
```

### 5.16.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Traversee the AST
2. If a cast expression casts away constness report an error.

### 5.16.6 References

ALE3D Style Guide section 14.4.

## 5.17 Constructor Destructor Calls Virtual Function

*C++ Coding Standards*, states that:

> Virtual functions only "virtually" always behave virtually: Inside constructors and destructrors, they don't. Worse, any direct or indirect call to an unimplemented *pure virtual* function from a constructor or destructor results in undefined behavior. If your design wants virtual dispatch into a derived class from a base class constructor or destructor, you need other techniques such as post-constructors.

### 5.17.1 Parameter Requirements

This checker takes no parameters and inputs source file

### 5.17.2 Implementation

This pattern is checked using a nested AST traversal in which the top level traversal seeks out definitions of constructors and destructors and two nested traversals seek out calls to virtual functions of member functions and non-member functions respectively.

### 5.17.3 Non-Compliant Code Example

The following code calls a virtual function from a public constructor. This is a contrived trivial example.

```
class Class
{
  int n;

  public:
    Class() { n = Classy(); } //constructor
    ~Class() {} //Destructor

    virtual int Classy() { return 1; }
}; //class Class

int main()
{
  Class c;
  return 0;
} //main()
```

### 5.17.4 Compliant Solution

```
class Class
{
```

```
  int n;

  public:
    Class() { n = 1; } //constructor
    ~Class() {} //Destructor
}; //class Class

int main()
{
  Class c;
  return 0;
} //main()
```

### 5.17.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform a AST traversal visiting class constructors and destructors.
2. Flag any calls to virtual functions in class constructor or destructor nodes.
3. Report any violations.

### 5.17.6 References

Alexandrescu A. and Sutter H. *C++ Coding Standards 101 Rules, Guidelines, and Best Practices*. Addison-Wesley 2005.

## 5.18 Secure Coding : STR05-A. Prefer making string literals const-qualified

String literals are constant and should consequently be protected by the `const` qualification. This recommendation supports rule STR30-C. Do not attempt to modify string literals .

### 5.18.1 Non-Compliant Code Example

In the following non-compliant code, the `const` keyword has been omitted.

```
1 char *c = "Hello";
2
```

If a statement such as `c[0] = 'C'` were placed following the above declaration, the code would likely still compile cleanly, but the result of the assignment is undefined as string literals are considered constant.

### 5.18.2 Compliant Solution 1

In this compliant solution, the characters referred to by the pointer `c` are `const`-qualified, meaning that any attempts to assign them to different values is an error.

```
1 char const *c = "Hello";
2
```

### 5.18.3 Compliant Solution 2

In cases where the string is meant to be modified, use initialization instead of assignment. In this compliant solution, `c` is a modifiable `char` array which has been initialized using the contents of the corresponding string literal.

```
1 char c[] = "Hello";
2
```

Thus, a statement such as `c[0] = 'C'` is valid and will do what is expected.

### 5.18.4 Non-Compliant Code Example 1

Although this code example is not compliant with the C99 Standard, it executes correctly if the contents of `CMUfullname` are not modified.

```
1 char *CMUfullname = "Carnegie Mellon University";
2 char *school;
3
4 /* Get school from user input and validate */
5
6 if (strcmp(school, "CMU")) {
7     school = CMUfullname;
8 }
9
```

### 5.18.5 Non-Compliant Code Example 2

Adding in the `const` keyword will likely generate a compiler warning, as the assignment of `CMUfullname` to `school` discards the `const` qualifier. Any modifications to the contents of `school` after this assignment will lead to errors.

```
1 char const *CMUfullname = "Carnegie Mellon University";
2 char *school;
3
4 /* Get school from user input and validate */
5
6 if (strcmp(school, "CMU")) {
7     school = CMUfullname;
8 }
9
```

### 5.18.6 Compliant Solution

The compliant solution uses the `const` keyword to protect the string literal, as well as using `strcpy()` to copy the value of `CMUfullname` into `school`, allowing future modification of `school`.

```
1 char const *CMUfullname = "Carnegie Mellon University";
2 char *school;
3
4 /* Get school from user input and validate */
5
6 if (strcmp(school, "CMU")) {
7     /* Allocate correct amount of space for copy */
8     strcpy(school, CMUfullname);
9 }
10
```

### 5.18.7 Risk Assessment

Modifying string literals causes undefined behavior, resulting in abnormal program termination and denial-of-service vulnerabilities.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---|---|---|---|---|---|
| STR05-A | **1** (low) | **3** (likely) | **2** (medium) | **P6** | **L3** |

#### Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website .

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1993/N0389.asc [ ISO/IEC 9899-1999:TC2 ] Section 6.7.8, "Initialization" [Lockheed Martin 2005] Lockheed Martin. Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program. Document Number 2RDU00001, Rev C. December 2005. AV Rule 151.1

## 5.19 Control Variable Test Against Function

This checker detects if there exists a for loop that tests its control (induction) variable against a function. One can get better performance by pulling out the function call before the loop and use a constant value for the test. That is,

```
for(int i = 0; i < constSize(); ++i)
{ // do something }
```

The code above can be improved as the following:

```
const int size = constSize();
for(int i = 0; i < size; ++i)
{ // do something }
```

### 5.19.1 Parameter Requirements

None

### 5.19.2 Implementation

This checker uses a simple traversal. For every `for` statement, the checker examines whether or not there is a function call inside the test expression.

### 5.19.3 Non-Compliant Code Example

```
int bar();

void foo()
{
  int j=2;

  for(int i = 0; i < bar(), j < 10; ++i)
  {
    j += 2;
  }
}
```

### 5.19.4 Compliant Solution

```
int bar();

void foo()
{
  int j=2;

  for(int i = 0; i < 10; ++i)
  {
    j += 3;
  }
}
```

### 5.19.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Check if a node is a for statement
2. Check if the test expression for the for statement contains a function call

### 5.19.6 References

The Programming Research Group, High-Integrity C++ Coding Standard Manual, Item 5.7: "The control variable in a for loop should be tested against a constant value, not a function"

## 5.20 Copy Constructor Const Arg

This checks whether the copy constructor for a class uses a const reference as an argument. This should always be the case as a copy constructor should never change its input argument and a reference is necessary to avoid needing a copy operator.

### 5.20.1 Parameter Requirements

No Parameter necessary.

### 5.20.2 Implementation

This checker begins by finding class declarations and getting the class name. It then runs through each member of the class until finding a constructor. It checks if the constructor has one argument and that argument is a member of the same class. If it is and it is not const a message is returned.

### 5.20.3 Non-Compliant Code Example

```
class interviewer
{
interviewer(interviewer other) {return;}
}
```

### 5.20.4 Compliant Solution

```
class interviewer
{
interviewer(const interviewer& other) {return;}
}
```

### 5.20.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Identify member function
2. check args for copy constructor
3. ensure type and const
4. if not both, return notification

### 5.20.6 References

Abbreviated Code Inspection Checklist Section 12.1.2, Copy Constructor"

## 5.21 Cpp Calls Setjmp Longjmp

*The Elements of C++ Style* state that:

> [The setjmp() and longjmp()] functions provide exception handling for C programs. You cannot safely use these functions in C++ code because the exception-handling mechanism they implement does not respect normal object lifecycle semantics–a jump will not result in destruction of scoped, automatically allocated objects.

### 5.21.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.21.2 Implementation

This pattern is checked using a simple AST traversal that seeks out calls to setjmp() and longjmp() in source files without the ".c" extension. These nodes are flagged as violations.

### 5.21.3 Non-Compliant Code Example

This contrived trivial example calls setjmp() and longjmp() in C++ code

```
#include <setjmp.h>

int main()
{
  jmp_buf env;
  my_container c1;
  int i = setjmp(env);

  if( i != 0 ) exit( 1 );

  int err = c1.clear();

  if( err != 0 ) longjmp( env, 1 );

  return 0;
}
```

### 5.21.4 Compliant Solution

The compliant solution uses C++ exception handling.

```
int main()
{
  my_container c1;

  try
```

```
  {
    c1.clear();
  }
  catch(...)
  {
    exit( 1 );
  }

  return 0;
}
```

### 5.21.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal visiting function reference codes.
2. Flag all function references named `setjmp()` or `longjmp()` as violations.
3. Report all violations.

### 5.21.6 References

Bumgardner G., Gray A., and Misfeldt T. *The Elements of C++ Style*. Cambridge University Press 2004.

## 5.22 Cycle Detection

This checker is a graph based checker. It determines if cycles are present in the crontrol flow graph.

### 5.22.1 Parameter Requirements

Parameters for the run must be provided, e.g. is the analysis interprocedural? This is work in progress.

### 5.22.2 Implementation

Within the traversal of the graph, the run function of this checker is called. This function takes the current and previous node as input. Within the run function, the algorithm checks the successors of the current node and determines if a node has been seen before during the graph traversal. If so, a possible cycle has been found. To verify that a cycle exists, we verify that a path from the successor to the current node exists.

### 5.22.3 Non-Compliant Code Example

```
% write your non-compliant code example
```

### 5.22.4 Compliant Solution

```
% write your compliant code example
```

### 5.22.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Write your checker algorithm

### 5.22.6 References

## 5.23 Paper: Cyclomatic Complexity

This is a checker to detect functions with high complexity. High complexity is defined by Mc Cabe's cyclomatic complexity metric. This metric measures the amount of branches in a function, i.e. through if and switch conditions.

### 5.23.1 Non-Compliant Code Examples

```
void fail() {
  int x;
  x=5;
  if (x>3) {
  }
  if (x>3) {
  }
  if (x>3) {
  }
}
```

### 5.23.2 Compliant Solution

```
void pass() {
  int x;
  x=5;
  if (x>3) {
  }
}
```

### 5.23.3 Parameter Requirements

CyclomaticComplexity.maxComplexity defines the max value for the complexity analysis.

### 5.23.4 Implementation

The algorithm searches for each function the number of occurances of:

```
  if (isSgIfStmt(node) || isSgCaseOptionStmt(node) || isSgForStatement(node) || isSgDo
    complexity++;
  }
```

### 5.23.5 References

McCabe , "Thomas McCabe, A Complexity Measure - IEEE Transactions on Software Engineering, Vol SE-2, No.4, December 1976."

## 5.24 Data Member Access

Following the spirit of data hiding in object-oriented programming, classes should in general not have public data members as these might give away details of the underlying implementation, making a change of implementation more difficult and possibly giving users a way to mess up the internal state of objects of the class. It is, however, sometimes useful to have 'behaviorless aggregates', i. e. C-style structs where all data members are public (and no member functions are present).

This checker warns about class definitions that fit into neither of the above patterns. Specifically, it warns about class definitions that contain both public and nonpublic data members.

### 5.24.1 Parameter Requirements

This checker does not require any parameters.

### 5.24.2 Non-Compliant Code Example

```
class NoNo { // not acceptable, contains both public and nonpublic data members
public:
    int get_a() const;
    void set_a(int);

    double d;

protected:
    int a;
};
```

### 5.24.3 Compliant Solution

```
struct C_style { // acceptable, all data members are public
    int a, b;
    double d;
};

class WellProtected { // acceptable, no data members are public
public:
    int get_a() const;
    void set_a(int);

    double get_d() const;
    void set_d(double);

protected:
    int a;

private:
```

```
    double d;
};
```

### 5.24.4 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each class definition, count the numbers of public and nonpublic data members.
2. If both of the counts for a given class definition are greater than zero, a mix of public and nonpublic data members is present; emit a diagnostic.

### 5.24.5 References

A literature reference for this checker is: H. Sutter, A. Alexandrescu: "C++ Coding Standards"", Item 41: "Make data members private, except in behaviorless aggregates (C-style structs)". Note that the authors advise not only against public, but also against protected data members; this checker does not report protected data.

## 5.25 Deep Nesting

It is widely agreed that functions should not be 'too big' (without a good reason, at least). Various size measures exist, including source code lines and cyclomatic complexity. This checker adds one more: It tests if function definitions contain more nested scopes than allowed by the user.

### 5.25.1 Parameter Requirements

This checker requires an integer entry DeepNestingChecker.maximumNestedScopes in the Compass parameters specifying the maximum allowed number of nested scopes.

### 5.25.2 Non-Compliant Code Example

```
/* The innermost scope (the if statement) in this toy function will be
 * reported by the DeepNestingChecker if maximumNestedScopes is set to 2. */
void matrix_abs(int n, int m, int **matrix)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (matrix[i][j] < 0)
            {
                matrix[i][j] = -matrix[i][j];
            }
        }
    }
}
```

### 5.25.3 Compliant Solution

```
/* The nesting in each function is not greater than 2; the if statement has
 * been pulled out into its own function. */
void abs_if_necessary(int *p)
{
    if (*p < 0)
    {
        *p = -*p;
    }
}

void matrix_abs2(int n, int m, int **matrix)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
```

```
            abs_if_necessary(&matrix[i][j]);
        }
    }
}
```

### 5.25.4 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each 'scope statement' (loops, if, basic blocks) count the number of enclosing scopes until a function definition is reached (if at all).
2. If the count is greater than the specified limit, emit a diagnostic.

### 5.25.5 References

A reference for this checker is: H. Sutter, A. Alexandrescu: "C++ Coding Standards", Item 20: "Avoid long functions. Avoid deep nesting".

## 5.26 Default Case

This test checks to ensure each switch statement has a default option. It has been noted that unexpected cases 'falling through" can be a cause of difficult to detect bugs. A default case will catch those cases.

### 5.26.1 Parameter Requirements

No parameters required.

### 5.26.2 Implementation

This implementation checks all statements in the basic block of the switch statement, searching for defaults. Currently this implementation may have false positives (ie. A switch with a default will raise an alert) on duff's device. In general Duff's device does not use a default.

### 5.26.3 Non-Compliant Code Example

```
switch(x)
{
case 1:
...
case 2:
...
}
```

### 5.26.4 Compliant Solution

```
switch(x)
{
case 1:
...
case 2:
...
default:
//handle unexpected cases
}
```

### 5.26.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Identifies switch statement

2. Reads all statements in it's basic block searching for default
3. if none found, notify message.

### 5.26.6 References

Abbreviated Code Inspection Checklist Section 11.2.2, Branching"

## 5.27 Default Constructor

*The Elements of C++ Style* item #97 states that

> Declare a default constructor for every class you create. Although some compilers may be able to automatically generate a more efficient implementation in some situations, choose an explicit default constructor for added clarity.

### 5.27.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.27.2 Implementation

This pattern is checked using a simple AST traversal that seeks out instances of SgClassDefinition. The children of these instances are stored in a successor container and looped over to find a default constructor. If no such default constructor exists then a violation is flagged.

### 5.27.3 Non-Compliant Code Example

The following trivial example does not declare a default constructor.

```
class Class
{
  public:
    ~Class(){}
}; //class Class
```

### 5.27.4 Compliant Solution

The compliant solution simply adds a default constructor to the class definition.

```
class Class
{
  public:
    Class(){}
    ~Class(){}
}; //class bad
```

### 5.27.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform AST traversal visiting the member functions of class definitions.
2. If no constructor is found for a single class definition then flag violation.
3. Report any violations.

### 5.27.6 References

Bumgardner G., Gray A., and Misfeldt T. *The Elements of C++ Style*. Cambridge University Press 2004

## 5.28 Discard Assignment

According to some coding standards, the assignment operator should not be used within larger constructs, but only as a stand-alone expression statement; in particular, it should not be used as the controlling expression in a branch because it might be confused with the equality operator. This checker reports any use of the assignment operator (built-in or overloaded) that is not the sole expression in an expression statement.

### 5.28.1 Parameter Requirements

This checker does not require any parameters.

### 5.28.2 Non-Compliant Code Example

```
void strcpy_noncompliant(char *dest, const char *source)
{
    while (*dest++ = *source++)
        ;
}
```

### 5.28.3 Compliant Solution

```
void strcpy_compliant(char *dest, const char *source)
{
    char last = *source;
    do {
        last = *source;
        *dest++ = *source++;
    } while (last != '\0');
}
```

### 5.28.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each assignment, geneate a diagnostic if its parent is not an expression statement.
2. For each assignment that has an expression statement as its parent, generate a diagnostic if that expression statement is the controlling expression statement of a loop, if, or switch.

### 5.28.5 References

A reference to this pattern is: The Programming Research Group: "High-Integrity C++ Coding Standard Manual", Item 10.5: "Always discard the result of an assignment operator."

## 5.29 Do Not Call Putenv With Auto Var

The POSIX function putenv() is used to set environment variable values. The putenv() function does not create a copy of the string supplied to it as an argument, rather it inserts a pointer to the string into the environment array. If a pointer to a buffer of automatic storage duration is supplied as an argument to putenv(), the memory allocated for that buffer may be overwritten when the containing function returns and stack memory is recycled. This behavior is noted in the Open Group Base Specifications Issue 6 [Open Group 04]:

A potential error is to call putenv() with an automatic variable as the argument, then return from the calling function while string is still part of the environment.

The actual problem occurs when passing a pointer to an automatic variable to putenv(). An automatic pointer to a static buffer would work as intended.

### 5.29.1 Parameter Requirements

No Parameter specifications.

### 5.29.2 Implementation

The putenv() function is not required to be thread-safe, and the one in libc4, libc5 and glibc2.0 is not, but the glibc2.1 version is.

Description for libc4, libc5, glibc: If the argument string is of the form name, and does not contain an '=' character, then the variable name is removed from the environment. If putenv() has to allocate a new array environ, and the previous array was also allocated by putenv(), then it will be freed. In no case will the old storage associated to the environment variable itself be freed.

The libc4 and libc5 and glibc 2.1.2 versions conform to SUSv2: the pointer argument given to putenv() is used. In particular, this string becomes part of the environment; changing it later will change the environment. (Thus, it is an error to call putenv() with a pointer to a buffer of automatic storage duration as the argument, then return from the calling function while the string is still part of the environment.) However, glibc 2.0-2.1.1 differs: a copy of the string is used. On the one hand this causes a memory leak, and on the other hand it violates SUSv2. This has been fixed in glibc2.1.2.

The BSD4.4 version, like glibc 2.0, uses a copy.

SUSv2 removes the 'const' from the prototype, and so does glibc 2.1.3.

The FreeBSD implementation of putenv() copies the value of the provided string, and the old values remain accessible indefinitely. As a result, a second call to putenv() assigning a differently sized value to the same name results in a memory leak.

### 5.29.3 Non-Compliant Code Example

In this non-compliant coding example, a pointer to a buffer of automatic storage duration is used as an argument to putenv() [Dowd 06]. The TEST environment variable may take on an unintended value if it is accessed once func() has returned and the stack frame containing env has been recycled.

Note that this example also violates rule [DCL30-C. Declare objects with appropriate storage durations].

```
int func(char *var) {
  char env[1024];

  if (snprintf(env, sizeof(env),"TEST=%s", var) < 0) {
    /* Handle Error */
  }

  return putenv(env);
}
```

### 5.29.4 Compliant Solution

The setenv() function allocates heap memory for environment variables. This eliminates the possibility of accessing volatile, stack memory.

```
int func(char *var) {
  return setenv("TEST", var, 1);
}
```

### 5.29.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Checks to see if the variable being passed to putenv() is declared in the global scope. If it is not, the checker creates new output.

### 5.29.6 References

Open Group 04 The putenv() function

ISO/IEC9899-1999Section 6.2.4, "Storage durations of objects," and Section 7.20.3, "Memory management functions"

Dowd 06 Chapter 10, "UNIX Processes" (Confusing putenv() and setenv())

## 5.30 Do Not Delete This

"CERT Secure Coding C++ DAN32-C." states that

> Deleting this leaves it as a "dangling" pointer, which leads to undefined behavior if it is accessed.

### 5.30.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.30.2 Implementation

This pattern is checked using a simple AST traversal visiting all `delete` expressions and checking its argument to be a `this` expression; if so, flag a violation.

### 5.30.3 Non-Compliant Code Example

```
class SomeClass {
    public:
        SomeClass();
        void doSomething();
        void destroy();
        // ...
};

void SomeClass::destroy() {
    delete this;  // Dangerous!!
}

SomeClass sc = new SomeClass;
// ...
sc->destroy();
// ...
sc->soSomething();  // Undefined behavior
```

### 5.30.4 Compliant Solution

```
class SomeClass {
    public:
        SomeClass();
        void doSomething();
        // ...
        ~SomeClass();
};

SomeClass sc = new SomeClass;
// ...
delete sc;
```

### 5.30.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal visiting all `delete` expression nodes.
2. For each `delete` expression node, check its argument node to be `this` expression; if so, flag violation.
3. Report any violations.

### 5.30.6 References

DAN32-C. Do not delete this

## 5.31 Do Not Use C-style Casts

C++ allows C-style casts, although it has introduced its own casts:

- static_cast<type>(expression)
- const_cast<type>(expression)
- dynamic_cast<type>(expression)
- reinterpret_cast<type>(expression)

C++ casts allow for more compiler checking and are easier to find in source code (either by tools or by human readers).

### 5.31.1 Parameter Requirements

No Parameter specifications.

### 5.31.2 Implementation

### 5.31.3 Non-Compliant Code Example

In this example, a C-style cast is used to convert an int to a double:

```
% write your non-compliant code example
int dividend, divisor;
// ...
double result = ((double)dividend)/divisor;
```

### 5.31.4 Compliant Solution

Using the new cast, the division should be written as:

```
% write your compliant code example
double result = static_cast<double>(dividend)/divisor;
```

### 5.31.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Check to see if the SgCastExp node is of type SgCastExp::e_C_style_cast, and if it is, add ouput.

### 5.31.6 References

Dewhurst 03 Gotcha 40: Old-Style Casts

ISO/IEC 14882-2003 Sections 5.2.9, 5.2.11, 5.2.7, 5.2.10.

Meyers 96 Item 2: Prefer C++-style casts.

Lockheed Martin 05 AV Rule 185 C++ style casts (const_cast, reinterpret_cast, and static_cast) shall be used instead of the traditional C-style casts.

## 5.32 Duffs Device

This test checks for the presence of Duff's Device in the source code. Duff's Device is a switch statement containing a loop (for, while or do-while; we do not check for goto loops) that contains one of the switch's case or default labels. If such a construct is found, the position of the switch statement is reported.

### 5.32.1 Parameter Requirements

This checker does not require any parameters.

### 5.32.2 Non-Compliant Code Example

```
// Duff's Device in its almost original form.
void send(int *to, int *from, int count)
{
  int n = (count+7) / 8;
  switch(count%8) {
  case 0: do { *to++ = *from++;
  case 7:      *to++ = *from++;
  case 6:      *to++ = *from++;
  case 5:      *to++ = *from++;
  case 4:      *to++ = *from++;
  case 3:      *to++ = *from++;
  case 2:      *to++ = *from++;
  case 1:      *to++ = *from++;
             } while (--n>0);
  }
}
```

### 5.32.3 Compliant Solution

```
// An equivalent function without optimization.
void send2(int *to, int *from, int count)
{
    for (int i = 0; i < count; i++)
        *to++ = *from++;
}
```

### 5.32.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each case or default statement, examine the enclosing scopes, smallest first (i.e. go upward in the AST).
2. If there is a loop statement that is closer than a switch statement, Duff's Device has been found; emit a diagnostic.

### 5.32.5 References

Duff's Device ist mostly folklore, but one reference in the literature is: B. Stroustrup: "The C++ Programming Language, Third Edition", Exercise §6.6[15].

## 5.33 Dynamic Cast

The rule is that dynamic_cast¡¿ should always be used when a downcast is needed (ALE3D 14.5). Downcasting is a term for casting a pointer to a base class to a derived class, and should only be done with extreme care. Using dynamic_cast¡¿ will ensure that the object pointed to by the base class pointer is really of the derived class. It will return a nill pointer if it is not. Any other type of cast is unsafe.

### 5.33.1 Parameter Requirements

This checker does not take any paramaters.

### 5.33.2 Implementation

This pattern is detected using a simple traversal without inherited or synthesized attributes.

### 5.33.3 Non-Compliant Code Example

```
class A//not polymorphic
 {
   public:
     ~A(){}
     virtual void foo(){};
 };

class B: public A
 {
 };

int main()
  {
    A * p  = new B;
    B * p2 = (B*) p;
  }
```

### 5.33.4 Compliant Solution

```
class A//not polymorphic
 {
   public:
     ~A(){}
     virtual void foo(){};
 };
```

```
class B: public A
 {
 };

int main()
  {
    A * p = new B;
    B * p2 = dynamic_cast<B* > (p);
  }
```

### 5.33.5 Mitigation Strategies

**Static Analysis**

1. traverse AST
2. for each cast expression that is a downcast if dynamic cast is not used report an error.

### 5.33.6 References

## 5.34 Empty Instead Of Size

While comparing the result of the size() member function on STL containers against 0 is functionally equivalent to calling the empty() member function, empty() is to be preferred as it is always a constant-time operation, while size() on std::list may take linear time. This checker detects cases where the result of size() is compared against the constant 0.

### 5.34.1 Parameter Requirements

This checker does not require any parameters.

### 5.34.2 Non-Compliant Code Example

```
#include <vector>

bool f(const std::vector<int> &v)
{
    if (v.size() > 0) // not OK: use !v.empty() instead
        return true;
    if (0 == v.size()) // not OK: use v.empty() instead
        return false;
    return false;
}
```

### 5.34.3 Compliant Solution

```
#include <vector>
bool f2(const std::vector<int> &v)
{
    if (!v.empty())
        return true;
    if (v.empty())
        return false;
    return false;
}
```

### 5.34.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each member function call, see if the called member function is named 'size' and if the call is embedded in an expression that compares its return value against the constant 0.

2. If the above check evaluates to true, emit a diagnostic.

There are numerous ways to defeat this simple analysis, for instance by assigning the return value from size() to a variable, by comparing the return value against a variable that is always 0, or by calling size() through a member function pointer. Further, the analysis only looks for member functions named 'size' but does not try to ascertain that it belongs to a 'container' (as that is not something that can be checked reliably).

### 5.34.5 References

The reference for this checker is: S. Meyers: "Effective STL", Item 3: "Call `empty` instead of checking `size()` against zero."

## 5.35 Enum Declaration Namespace Class Scope

*The Elements of C++ Style* item #79 states that

> To avoid symbolic name conflicts between enumerators and other global names, nest enum declarations within the most closely related class or common namespace.

### 5.35.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.35.2 Implementation

This pattern is checked using a simple AST traversal that locates nodes that are enumeration declarations. If a enumeration declaration is found then its parent nodes are traversed until a class or namespace declaration is found. If no namespace or class declaration(s) are found then a violation is flagged by this checker.

### 5.35.3 Non-Compliant Code Example

This non-compliant code contains an enum declaration at the global scope.

```
enum violation{ E1=0, E2, E3 }; // This is a violation
```

### 5.35.4 Compliant Solution

The compliant solution simply nests the violation enum declaration in a unique namespace.

```
namespace Namespace
{
  enum violation{ E1=0, E2, E3 }; // This is OK
} //namespace Namespace
```

### 5.35.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform an AST traversal visiting enum declaration nodes.
2. For each enum declaration node visit its parents checking them to be either namespace declarations or class declarations. If no class or namespace declaration parent node is found, then flag violation.
3. Report any violations.

### 5.35.6 References

Bumgardner G., Gray A., and Misfeldt T. *The Elements of C++ Style*. Cambridge University Press 2004.

## 5.36 CERT-DCL04-A: Explicit Char Sign

"CERT Secure Coding INT07-A" states

> The three types `char`, `signed char`, and `unsigned char` are collectively called the character types. Compilers have the latitude to define `char` to have the same range, representation, and behavior as either `signed char` or `unsigned char`. Irrespective of the choice made, `char` is a separate type from the other two and is **not** compatible with either.

### 5.36.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.36.2 Implementation

This pattern is checked using a simple AST traversal visiting all SgAssignInitializer nodes. If the SgAssignInitializer node is of type `char` and the operand of the node is a SgCastExp of type `int` or is a SgCharVal whose value is negative then flag an error.

### 5.36.3 Non-Compliant Code Example

This non-compliant example declares a simple `char` type variable.

```
#include <stdio.h>

int main()
{
  int n = 200;
  char c1 = 'i';
  char c2 = n;
  char c3 = 200;
  int i = 1000;

  printf( "%c/c2 = %d\n%c/c3 = %d\n", c1, i/c2, c1, i/c3);

  return 0;
}
```

### 5.36.4 Compliant Solution

The compliant solution explicitly declares the `char` variables as `unsigned`.

```
#include <stdio.h>

int main()
{
  int n = 200;
```

```
  char c1 = 'i';
  unsigned char c2 = n;
  unsigned char c3 = 200;
  int i = 1000;

  printf( "%c/c2 = %d\n%c/c3 = %d\n", c1, i/c2, c1, i/c3);

  return 0;
}
```

### 5.36.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal visiting all SgAssignInitializer nodes.
2. For each SgAssignInitializer node of type `char`, if the rhs operand is either a cast expression with operand of type int or a negative character value then flag an error.
3. Report any violations.

### 5.36.6 References

Secure Coding : INT07-A. Explicitly specify signed or unsigned for character types

## 5.37 Explicit Copy

This test detects missing copy constructors and operators. In case the user wants to use the default ones then the class has to be annotated with a special comment. These comments should contain "`use default copy constructor`" or "`use default copy operator`".

This checker enforces the rule 53 from H. Sutter, A. Alexandrescu *C++ Coding Standards*: "Explicitly enable or disable copying".

### 5.37.1 Parameter Requirements

No parameter is required.

### 5.37.2 Non-Compliant Code Example

```
class A {
};
```

### 5.37.3 Compliant Solution

```
class A {
public:
  A(const A& other) {

  }
  A& operator=(const A& other) {
    return this;
  }
};
```

### 5.37.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For all class definitions, try to find a copy constructor and a copy operator, or user comments describing that the class should use the default ones.

### 5.37.5 References

Alexandrescu A. and Sutter H. *C++ Coding Standards 101 Rules, Guidelines, and Best Practices.* Addison-Wesley 2005.

## 5.38 Explicit Test For Non Boolean Value

This test examines all the test statements whether there is a statement that calls a function call returning a non-boolean value and that does not compare the return value to an explicit value. For example, if a function `foo()` returns an integer value and the function is used in a conditional statement, such as `"if", "while", "do-while", "for"`, or the first operand of "?" operator, the boolean expressions in the conditional statement should always use an explicit test of equality or non-equality. Therefore the following code can pass this checker:

```
if(foo()!=0)
{// do something}
```

whereas,

```
if(foo())
{// do something}
```

will be caught by this checker because `foo()` returns an integer, non-boolean value.

### 5.38.1 Parameter Requirements

None

### 5.38.2 Implementation

This pattern is detected using a simple traversal. It traverses AST to search conditional statements and if an implicit expression is used in the test, AST contains a casting expression node underneath the conditional statement to convert from a non-boolean values to a boolean value. The checker captures this structure.

### 5.38.3 Non-Compliant Code Example

```
int bar();

void foo()
{
  int i;
  if(bar())
    i = 2;

  while(bar())
    i = 3;

  do {
    i = 4;
  } while(bar());
```

```
  for(i=0; bar(); i++)
    i =5;

  i = (bar() ? 6 : 7);

  for(i = (bar() ? 8 : 9); bar(); i++)
    i = 10;
}
```

### 5.38.4 Compliant Solution

```
if(foo()!=0)
{// do something}
```

### 5.38.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Check if a node is a conditional statement
2. Check further if the conditional statement contains an implicit expression.

### 5.38.6 References

The Programming Research Group, High-Integrity C++ Coding Standard Manual, Item 5.2: "For boolean epressions('if', 'for', 'while', 'do' and the first operand of the ternary operator '?:') involving non-boolean values, always use an explicit test of equality or non-equality."

## 5.39 File Read Only Access

Attempting to open files for writing on read-only file systems and files causes errors. This checker checks that all standard C/C++ file I/O is read-only. This checker is normally used only internally for compass verify.

### 5.39.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.39.2 Implementation

This pattern is checked using a simple AST traversal visiting all function call expressions calling the function `fopen`. The mode parameter to `fopen` is checked to be either "r" or "rb" only. Any parameter that violates this restriction is flagged as an error. Additionally, all variable declarations of type `fstream` and `ofstream` are forbidden by this checker.

### 5.39.3 Non-Compliant Code Example

```
#include <fstream>
#include <stdio.h>

int main()
{
  FILE *f1 = fopen( "f1.txt", "w" );
  std::fstream f2( "f2.txt", std::ios::app | std::ios::out );

  FILE *f1r = fopen( "f1.txt", "r" );
  std::ifstream f2r( "f2.txt", std::ios::in );

  return 0;
}
```

### 5.39.4 Compliant Solution

Do not write to file.

### 5.39.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal on all function call expressions calling the `fopen` function.
2. Check the mode parameter of `fopen` to be either "r" or "rb"; if not, then flag violation.

3. Perform simple AST traversal on all variable declarations of the type `fstream` or `ofstream`. Forbid the use of these declarations and mark them as violations.

4. Report all violations.

### 5.39.6 References

## 5.40 Float For Loop Counter

"CERT Secure Coding" states

> Floating point arithmetic is inexact and is subject to rounding errors. Hence, floating point variables should not be used as loop counters.

### 5.40.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.40.2 Implementation

This pattern is checked using a simple AST traversal that visits all for loop init statement nodes and checks the type of its counter variable declaration. If that type is `float` or `double` then flag a violation.

### 5.40.3 Non-Compliant Code Example

```
for (float count = 0.1f; count <= 1; count += 0.1f)
{
}
```

### 5.40.4 Compliant Solution

The compliant solution uses an `int` type loop counter.

```
for (int count = 1; count <= 10; count += 1)
{
}
```

### 5.40.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal visiting all for loop initialization statement nodes.
2. For each node check the type of its variable declaration. If type is `float` or `double` then flag violation.
3. Report any violations.

### 5.40.6 References

FLP31-C. Do not use floating point variables as loop counters

## 5.41 Floating Point Exact Comparison

This checker detects a test clause that compares a variable to a floating point value. The rationale for this checker is, floating point representations are platform dependent, so it is necessary to avoid exact comparisons.

### 5.41.1 Parameter Requirements

None.

### 5.41.2 Implementation

This checker is implemented with a simple AST traversal. It traverses AST and finds a test clause. If the test clause has a double value on either left-hand-side or right-hand-side, and if the operator used for the test is "==" or "!=", then the checker reports this clause.

### 5.41.3 Non-Compliant Code Example

```
void foo( double f )
{
if ( f == (float)3)
{
f = 1.234;
}

while(f != 1.23456)
{
f += 0.00001;
}

do
{
f += 0.000001;
} while ( f != 1.234567);

for(f = 1.234567; f != 1.2345678; f += 0.0000001)
{
int i = f + 1;
}
}
```

### 5.41.4 Compliant Solution

```
bool double_equal(const double a, const double b)
{
const bool equal = fabs(a-b) < numeric_limits<double>::epsilon;
return equal;
}
```

```
void foo(double f)
{
if(double_equal(f, 3.142))
{
// do something
}
}
```

### 5.41.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Check if a node is a test clause
2. Check further if the clause has a double value and if the test is for (in)equality.

### 5.41.6 References

The Programming Research Group, High-Integrity C++ Coding Standard Manual, "Item 10.15: Do not write code that expects floating point calculations to yield exact results".

## 5.42 Fopen Format Parameter

"CERT Secure Coding FIO11-A" states

> The C standard specifies specific strings to use for the `mode` for the function `fopen()`. An implementation may define extra strings that define additional modes, but only the modes in the following table (adapted from the C99 standard) are fully portable and C99 compliant:

1. r
2. w
3. a
4. rb
5. wb
6. ab
7. r+
8. w+
9. a+
10. r+b or rb+
11. w+b or wb+
12. a+b or ab+

### 5.42.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.42.2 Implementation

This pattern is checked using a simple AST traversal that visits all function call expressions. For each function call expression the function name is confirmed to be `fopen()` then the format parameter is checked against the list of specified strings. If the given parameter is not a standard format string then a violation is flagged.

### 5.42.3 Non-Compliant Code Example

```
#include <stdio.h>

int main()
{
  FILE *f = fopen( "/tmp/tmp.txt", "wr" );
```

```
  fclose( f );

  return 0;
}
```

### 5.42.4 Compliant Solution

The compliant solution uses the "r+" specified parameter string instead.

```
#include <stdio.h>

int main()
{
  FILE *f = fopen( "/tmp/tmp.txt", "r+" );

  fclose( f );

  return 0;
}
```

### 5.42.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal visiting all function call expression nodes.
2. For each function call expression node, unparse node string then to determine the function name and parse the format parameter.
3. Check the format parameter against list of standard values. If given format parameter does not conform to list of specified values, then flag violation.
4. Report any violations.

### 5.42.6 References

Secure Coding : FIO11-A. Take care when specifying the mode parameter of fopen()

## 5.43 Forbidden Functions

Many checks common to Compass center around the forbidden use of certain "dangerous" functions. This checker provides a way to forbid the use of those functions through the simple use of their name.

### 5.43.1 Parameter Requirements

The forbidden function checker can simultaneously look for any number of functions, either member or non-member. A set of parameters is used, named using a counter. Thus, the parameters of this checker have names of the form `ForbiddenFunctions.Function`$n$, for $n$ from zero to some limit. The forbidden function analysis checks each name in turn until one is missing. Thus, if you have parameters named `ForbiddenFunctions.Function0` and `ForbiddenFunctions.Function1` but no parameter named `ForbiddenFunctions.Function2`, the analysis will search for two functions. As a caution, if you skip a number, including zero, no larger numbers will be scanned: any functions specified after a skipped number will be ignored.

The format of a parameter is `white space, function name, white space, comma, reason`. Leading and trailing white space is allowed next to the function name, but any white space after the first comma will become part of the reason string. The function name is a fully qualified name, but the leading `::` to indicate the global scope may be omitted. Member functions are given with their class qualifications, just as they would be referred to when accessing a pointer to them. Choosing one overload from an overload set sharing the same name is not supported. The reason field is used to indicate why a particular function is forbidden; it may be any string (not containing a newline), and is printed out as part of the error message when the corresponding function is found. It is also possible to omit the comma and the reason field, leaving just the function name as the parameter; in this case, no reason will be given for the function's prohibition.

### 5.43.2 Implementation

This pattern is checked using a simple AST traversal visiting all SgFunctionCallExp nodes. For each node the name of the function being called is compared to those listed as forbidden functions. If a match is found between the function call name and the forbidden function name then flag an error.

### 5.43.3 Non-Compliant Code Example

In this example, it is assumed that the function `compass_forbidden_function_vfork` is forbidden by the parameter file.

```
//compass_forbidden_function_vfork() is a function meant to simulate a system
//routine. This makes no difference as an example of how forbiddenFunctions
//checks errors as it works only on function names. The reason we have used
//this example is to prevent compass from treating this test file as an error
//during 'make verify' in which compass checks its own source code.
```

```
int compass_forbidden_function_vfork();
void helloWorld(int, char**);

double function();
double good_function();

namespace A {
  double function2();
}

struct B {
  void memberFunction();
};

int main(int argc, char** argv) {
  int w;
  w = compass_forbidden_function_vfork() + 5;
  helloWorld(argc, argv);
  double x = 3.0 * function();
  double y = 5.0 * good_function();
  double z = A::function2();
  B b;
  b.memberFunction();
  int (*fp)() = compass_forbidden_function_vfork;
  return 0;
}
```

### 5.43.4 Compliant Solution

The compliant example would use a function not listed in the parameter file.

### 5.43.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each SgFunctionCallExp node if the name of the function being called is forbidden by the parameter file then report error.

### 5.43.6 References

Foster , "James C.Foster, Vitaly Osipov, Nish Bhalla, Niels Heinen, Buffer Overflow Attacks, ISBN 1-932266-67-4, p. 211"

Secure Coding : MSC30-C. Do not use the rand function

Secure Coding : POS33-C. Do not use vfork()

[ ISO/IEC 9899-1999:TC2 ] Section 7.19.9.2, "The `fseek` function"; 7.19.9.5, "The `rewind` function"

[ Klein 02 ] [
ISO/IEC 9899-1999 ] Section 7.20.1.4, "The strtol, strtoll, strtoul, and strtoull functions," Section 7.20.1.2, "The atoi, atol, and atoll functions," and Section 7.19.6.7, "The sscanf function"

## 5.44 For Loop Construction Control Stmt

"ALE3D Coding Standards & Style Guide" item #6.1 states that

> for() construction loops must only include statements that control the loop. In particular, for() loops must not initialize or increment/decrement variables not directly related to the loop control.

### 5.44.1 Parameter Requirements

This checker takes no parameters and inputs the source file.

### 5.44.2 Implementation

This pattern is checked using a simple AST traversal that seeks out for loop statement constructs. A list of variables set in the initialization block is generated. A list of variables set in the increment/decrement block is generated. Any variable in the increment/decrement list of variables must be in the initialization list of variables.

### 5.44.3 Non-Compliant Code Example

This non-compliant code initializes the array inside the for() control statement.

```
#include <stdlib.h>

int main()
{
  int *array = (int*)malloc( 100*sizeof(int) );
  int j=100;

  for( int i = 0; i < 100; i++, j-- ){
     array[i] = j;
  }

  free(array);
  return 0;
} //main()
```

### 5.44.4 Compliant Solution

The compliant solution simply moves the array initialization inside the for() loop body.

```
#include <stdlib.h>

int main()
{
  int *array = (int*)malloc( 100*sizeof(int) );
```

```
  for( int i = 0; i < 100; i++ ){
     array[i] = i;
  }

  free(array);
  return 0;
} //main()
```

### 5.44.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal visiting all for statements.
2. Generate a list of variables set in the initialization block.
3. Generate a list of variables set in the increment/decrement block.
4. Report any variable in the increment/decrement list of variables that are not in the initialization list of variables as a violation

### 5.44.6 References

Arrighi B., Neely R., Reus J. “ALE3D Coding Standards & Style Guide”, 2005.

## 5.45 For Loop Cpp Index Variable Declaration

"ALE3D Coding Standards & Style Guide" Item #6.2 states that

> C++ loop index variables should be declared in the loop statement. Declaration of a loop index variable in the first clause of the for() statement ensures that its scope is limited to the loop body.

### 5.45.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.45.2 Implementation

This pattern is checked using a simple AST traversal that seeks SgForInitStatements that have NULL declaration statements. These nodes are flagged as violations.

### 5.45.3 Non-Compliant Code Example

This non-compliant code declares the loop control variable outside the loop statement.

```
int main()
{
  int i = 0;
  for( i = 0; i < 100; i++ ){}

  return 0;
} //main()
```

### 5.45.4 Compliant Solution

The compliant solution declares the index variable inside the loop statement.

```
int main()
{
  for( int i = 0; i < 100; i++ ){}

  return 0;
} //main()
```

### 5.45.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal visiting all declaration statement nodes.

2. For each declaration statement node, if parent node is for loop initialization statement then flag violation.

3. Report any violations.

### 5.45.6 References

Arrighi B., Neely R., Reus J. "ALE3D Coding Standards & Style Guide", 2005.

## 5.46 Friend Declaration Modifier

*The Elements of C++ Style* item #96 states that

> Friend declarations are often indicative of poor design because they bypass access restrictions and hide dependencies between classes and functions.

### 5.46.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.46.2 Implementation

This pattern is checked with a simple AST traversal that seeks declaration statements and determines if any use the "friend" modifier keyword. Any declaration statements found with the "friend" modifier are flagged as violations.

### 5.46.3 Non-Compliant Code Example

This non-compliant example uses "friend" to access private data.

```
class Class
{
  int privateData;
  friend int foo( Class & c );

  public:
    Class(){ privateData=0; }
}; //class Class

int foo( Class & c )
{
  return c.privateData + 1;
} //foo( Class & c )
```

### 5.46.4 Compliant Solution

The compliant solution simply uses an accessor function instead.

```
class Class
{
  int privateData;

  public:
    Class(){ privateData=0; }
    int getPrivateData(){ return privateData; }
}; //class Class

int foo( Class & c )
```

```
{
  return c.getPrivateData() + 1;
} //foo( Class & c )
```

### 5.46.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal and visit all declaration statement nodes
2. For each declaration statement check the "friend" modifier. If "friend" modifier is set then flag violation.
3. Report any violations.

### 5.46.6 References

Bumgardner G., Gray A., and Misfeldt T. *The Elements of C++ Style*. Cambridge University Press 2004.

## 5.47 Function Call Allocates Multiple Resources

"CERT Secure Coding RES30-C" states

> Allocating more than one resource in a single statement could result in a memory leak, and this could lead to a denial-of-service attack.

### 5.47.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.47.2 Implementation

This pattern is checked using a simple AST traversal on each SgFunctionCallExp node. For each node get the expression list of its arguments and check if any such argument expressions are the `new` keyword. If the number of `new` expressions exceeds one then flag an error.

### 5.47.3 Non-Compliant Code Example

```
class A
{
};

class B
{
};

int foo( A *a, B *b )
{
  return 0;
}

int main()
{
  A *a = new A;
  B *b = new B;
  int i = foo( a, b ); //ok...

  return foo( new A, new B ); //bad
}
```

### 5.47.4 Compliant Solution

See the call to `foo` above.

### 5.47.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Traverse all SgFunctionCallExp nodes
2. For each node get the list of argument expressions
3. Count the number of `new` keyword argument expressions
4. If the number of `new` keyword argument expressions exceeds one then flag an error.
5. Report all violations.

### 5.47.6 References

RES30-C. Never allocate more than one resource in a single statement

## 5.48 CERT-DCL31-C: Function Definition Prototype

"CERT Secure Coding DCL31-C" states

> Functions should always be declared with the appropriate function prototype. If a function prototype is not available, the compiler cannot perform checks on the number and type of arguments being passed to functions. Argument type checking in C is only performed during compilation, and does not occur during linking, or dynamic loading.

### 5.48.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.48.2 Implementation

This pattern is checked using a simple AST traversal of function declaration nodes. For each function declaration node find the first non-defining function declaration; if none is found, then flag violation.

### 5.48.3 Non-Compliant Code Example

This example `foo()` has no prototype.

```
int foo( int i )
{
  return i;
}

int main()
{
  return foo(0);
}
```

### 5.48.4 Compliant Solution

The compliant solution simply adds a function prototype for `foo()`

```
int foo(int);

int foo( int i )
{
  return i;
}

int main()
{
  return foo(0);
}
```

### 5.48.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform a simple AST traversal of code visiting all function declaration nodes.
2. For each function declaration node, find first non-defining declaration. If no non-defining declaration is found, then flag violation.
3. Report any violations detected.

### 5.48.6 References

Secure Coding : DCL31-C. Ensure every function has a function prototype

## 5.49 Paper: Function Documentation

This analysis detects all non-compiler generated functions and checks whether they are documented. The documentation must be in front of the function.

### 5.49.1 Non-Compliant Code Examples

```
void fail() {
  //  this function has no comment in front of it
}
```

### 5.49.2 Compliant Solution

```
//Your test file code goes here.
void succeed() {
}
```

### 5.49.3 Parameter Requirements

None.

### 5.49.4 Implementation

This analysis checks for both '//' and '/*' documentation.

### 5.49.5 References

Panas05 , "Thomas Panas, Rudiger Lincke, Jonas Lundberg, Welf Lowe, A Qualitative Evaluation of a Software Development and Re-Engineering Project, NASA/IEEE Software Engineering Workshop, Washington DC, USA, April 2005"

## 5.50 Induction Variable Update

This test finds the location in loops (for, do-while, while) where induction variables is updated (through arithmetic operations).

### 5.50.1 Parameter Requirements

None.

### 5.50.2 Implementation

This pattern is detected using a simple traversal. It traverses AST to obtain information about induction variables and to locate statements that assign a new value to the induction variables. However, this checker does not track pointers whether or not the pointers actually update induction variables. In addition, function calls that may update induction variables are not considered here, either.

### 5.50.3 Non-Compliant Code Example

```
void foo(){
  int i;
  int j = 0;
  int k = 0;

  for(i = 0; i != 10; ++i)
  {
    if( 0 == i % 3)
    {
      i = 3;
      ++i;
      i++;
    }
  }

  while(j < 10)
  {
    if(1 == j %3)
    {
      j = j + 2;
    }
    j++;
  }

  do {
    if(2 == k % 3)
    {
      k +=1;
    }
```

```
  } while(++k < 10);
}
```

### 5.50.4 Compliant Solution

```
% write your compliant code example
```

### 5.50.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Find a loop and detect its induction variable
2. Check if the variable is updated inside the loop, by examining its loop body.

### 5.50.6 References

The Programming Research Group, High-Integrity C++ Coding Standard Manual, Item 5.6: "Do not alter a control variable more than once in a for, do or while statement."

## 5.51 Internal Data Sharing

Classes should usually not return handles to internal data from methods. A 'handle' in this sense is a non-const reference to a member or copy of a pointer member, as the caller could change internal state through such an object. This checker reports such cases. One possible exception to this rule are overloaded operators, which often return such handles (so they can be combined with other operators to bigger expressions), and this checker provides a parameter to define whether operators should be allowed to return internals.

### 5.51.1 Parameter Requirements

The bool flag **InternalDataSharing.operatorsExcepted** states whether overloaded operators are excepted from this checker's rules, i.e. whether they are allowed to return internal data.

### 5.51.2 Non-Compliant Code Example

```
class A
{
public:
    // not OK: returning non-const pointer member
    int *retptr() { return p; }
    // not OK: returning non-const reference to data pointed to by member
    int &retref() { return *p; }
    // not OK: returning non-const reference to member
    int *&retptrref() { return p; }

private:
    int *p;
};
```

### 5.51.3 Compliant Solution

```
class B
{
public:
    // OK: returning copy of pointed-to data
    int retint() { return *p; }
    // OK: const ptr
    const int *retcptr() { return p; }
    // OK: const ref
    const int &retcref() { return *p; }

    // maybe OK, depending on "operatorsExcepted" parameter
    int &operator*() { return *p; }

private:
    int *p;
```

```
};
```

### 5.51.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. While traversing the program representation, set a flag upon entering a member function definition that has a non-const pointer or reference return type.

2. Report any return statement within such a flagged function that returns a (possibly dereferenced) member variable.

### 5.51.5 References

A reference in the literature is: H. Sutter, A. Alexandrescu: "C++ Coding Standards", Item 42: "Don't give away your internals". Their notion of a handle is more general, however, as it also includes basic types that are used as handles, such as ints that are used as file descriptors.

## 5.52 Localized Variables

This checker looks for variable declarations and try to determine if the first use is far far the declaration. There can be two ways where the use is far. Either it is used only in an inner scope. Then the declaration can be moved in that scope. Or the it is used not directly after the block of declaration the variables is from. Then the declaration should be moved down.

This checker try to check for item 18 of *C++ Coding Standards* (Sutter and al., 2005).

### 5.52.1 Parameter Requirements

No parameter is needed.

### 5.52.2 Non-Compliant Code Example

```
void print(int);

void f()
{
  int i; //i should be declared at the for loop.
  int sum = 0; //sum is OK.

  //i is only used in the for scope.
  for (i = 0; i < 10; ++i) {
    //sum is used right after the block of declaration it belongs.
    sum += i;
  }

  //sum is used in the scope of definition.
  print(sum);
}
```

### 5.52.3 Compliant Solution

```
void print(int);

void f()
{
  int sum = 0;

  for (int i = 0; i < 10; ++i) {
    sum += i;
  }

  print(sum);
}
```

### 5.52.4 Mitigation Strategies

#### Static Analysis

The checker uses a scoped symbol table to track some properties about variable: Was the variable used? Was the variable in the same scope as its declaration? Was the variable used right after the the declaration? Is the declaration right before the current point of the traversal? The traversal updates these flags. Once a scope finished to be traversed, since the variables declared in the scope cannot be used any further, they are checked for the flags: used, used in the same scope and used right after its declaration.

There is another flag saying if the variable is a constant. In this case only check that the variable have been used, whereever it is.

This implementation does not care about aliasing. For more informations see subsection 5.52.6.

### 5.52.5 References

Alexandrescu A. and Sutter H. *C++ Coding Standards 101 Rules, Guidelines, and Best Practices*. Addison-Wesley 2005.

### 5.52.6 Limitations

This checker does not do alias analyzing. In the case you use a reference of a variable in an inner scope and re-use in the scope of declaration, the checker will not see the use and propose to move the variable in the inner scope. The effort for supporting the problem is very big, and the result is to handle programs with weird behavior that should have been written differently.

## 5.53 [No Reference] : Loc Per Function

This analysis detects for each function the amount of lines of code (LOC) and checks the value against a user defined max value. If LOC > max value, then an exception is triggered.

### 5.53.1 Non-Compliant Code Examples

```
// if LocPerFunction.Size = 2
void fail() {
  int x;
  x = 5;
  x = 5;
  x = 5;
}
```

### 5.53.2 Compliant Solution

```
// if LocPerFunction.Size = 2
void pass() {
  int x;
  x = 5;
  x = 5;
}
```

### 5.53.3 Parameter Requirements

LocPerFunction.Size defines the max value for a permissive LOC.

### 5.53.4 Implementation

The simple implementation of this checker is defined below:

```
  if (isSgFunctionDeclaration(sgNode)) {
    SgFunctionDeclaration* funcDecl = isSgFunctionDeclaration(sgNode);
    SgFunctionDefinition* funcDef = funcDecl->get_definition();
    if (funcDef) {
      Sg_File_Info* start = funcDef->get_body()->get_startOfConstruct();
      Sg_File_Info* end = funcDef->get_body()->get_endOfConstruct();
      ROSE_ASSERT(start);
      ROSE_ASSERT(end);
      int lineS = start->get_line();
      int lineE = end->get_line();
      loc_actual = lineE-lineS;
      if (loc_actual>loc) {
        output->addOutput(new CheckerOutput(funcDef));
      }
    }
  }
```

### 5.53.5 References

## 5.54 Lower Range Limit

By always using inclusive lower limits and exclusive upper limits, a whole class of off- by-one errors is eliminated. Furthermore, the following assumptions always apply: 1)the size of the interval equals the difference of the two
2) the limits are equal if the interval is empty
3) the upper limit is never less than the lower limit

Examples: instead of saying x¿=23 and x¡=42, use x¿=23 and x¡43.

### 5.54.1 Parameter Requirements

No parameters required.

### 5.54.2 Implementation

In a fairly straight-forward implementation we search the strictly lower than operator.

### 5.54.3 Non-Compliant Code Example

```
int x = 5;
if (x < 5)
{
x++;
}
% write your non-compliant code example
```

### 5.54.4 Compliant Solution

```
int x = 5;
if (x <= 4)
{
x++;
}
% write your compliant code example
```

### 5.54.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. find less than operator
2. raise alert

### 5.54.6 References

Abbreviated Code Inspection Checklist Section 11.1.1, Control Variables"

## 5.55 Magic Number

This test checks for the presence of 'magic numbers' in the source code. Magic numbers are all constants of integer or floating point type that occur outside of initializer expressions. The user may configure the checker to ignore certain common constants such as 0 or 1. This detector reports not only hand-written constants but also those that were created by macro expansion.

Note that C++ does not have negative constants; `-1` is not an integer constant but rather the unary minus operator applied to the constant `1`. To ignore occurrences of `-1`, you must therefore instruct the checker to ignore the constant `1`, but this will also ignore all 'positive' occurrences.

### 5.55.1 Parameter Requirements

The magic number detector requires two entries in the parameter file, one of which is the list of integer constants to ignore, the other the list of floating point constants to ignore. Constants in both lists are separated by whitespace; as explained above, it does not make sense to specify negative values.

An example of parameter entries is:

**MagicNumberDetector.allowedIntegers =**

**MagicNumberDetector.allowedFloats = 42.0 3.14159**

This specification has an empty list of integers, so every integer constant (of any type) will be flagged as a magic number; the floating point constants 42.0 and 3.14159 are allowed to appear in the source code, but all others are treated as magic numbers. Note that floating point numbers are compared by numeric value, which may result in strange effects due to inexact representation.

### 5.55.2 Non-Compliant Code Example

```
int f_noncompliant(int n)
{
    int x;
    x = 42; // not OK: magic number
    return x + n;
}
```

### 5.55.3 Compliant Solution

```
int f_compliant(int n)
{
    int x = 42; // OK: constant only used in initializer
    return x + n;
}
```

### 5.55.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For every integer or floating point literal, examine the enclosing statement to find out whether it occurs as part of the initializer in a variable declaration. If not, emit a diagnostic.

### 5.55.5 References

A reference for this pattern is: H. Sutter, A. Alexandrescu: "C++ Coding Standards", Item 17: "Avoid magic numbers".

## 5.56 Malloc Return Value Used In If Stmt

"ALE3D Coding Standards & Style Guide" item #4.5 states that

> When using raw `malloc()` and `new`, developers should check the return value for `NULL`. This is especially important when allocating large blocks of memory, which may exhaust heap resources.

### 5.56.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.56.2 Implementation

This pattern is checked using a simple AST traversal that seeks out function references to malloc. Then the parent nodes are traversed up until a basic scope block is found at which point a nested AST traversal seeks If-statement conditional expressions containing the memory block returned from malloc. If no such If-statement conditional is found in the immediate basic containing block scope then an error is flagged.

### 5.56.3 Non-Compliant Code Example

The non-compliant code fails to check the return value of `malloc()`.

```
#include <stdlib.h>

int main()
{
  int *iptr = (int*)malloc( 256*sizeof(int) );

  return 0;
} //main()
```

### 5.56.4 Compliant Solution

The compliant solution uses an if statement to check the return value of `malloc()` for `NULL`.

```
#include <stdlib.h>

int main()
{
  int *iptr = (int*)malloc( 256*sizeof(int) );

  if( iptr == NULL )
    return 1;

  return 0;
} //main()
```

### 5.56.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform AST traversal visiting function call nodes corresponding to `malloc()`.
2. For each call to `malloc()` traverse its parent nodes until an if statement or the end of a basic block is reached.
3. If an if statement is encountered, check that the if statement performs a comparison involving the return value from `malloc()`; if this is not the case then flag a violation.
4. If a basic block is reached, then flag a violation as the return value of `malloc()` may be out of scope.
5. Report any violations.

### 5.56.6 References

Arrighi B., Neely R., Reus J. “ALE3D Coding Standards & Style Guide”, 2005.

## 5.57 Multiple Public Inheritance

Multiple inheritance in C++ can give rise to very complicated issues, in particular when a class has several public superclasses; in contrast, having a single public superclass and several private ones (only inheriting code from these, but not public interfaces) can be much more controllable. This checker ensures that no class has more than one public superclass, while not prohibiting multiple inheritance in general.

### 5.57.1 Parameter Requirements

This checker does not require any parameters.

### 5.57.2 Non-Compliant Code Example

```
// Dummy classes, the first of which is designed to be used as a base class
// from which one inherits an interface, the second designed to be used as a
// base class from which one only inherits an implementation.
class Interface { /* ... */ };
class Implementation { /* ... */ };

// not OK: multiple public base classes
class A: public Interface, public Implementation
{
    /* ... */
};
```

### 5.57.3 Compliant Solution

```
class Interface { /* ... */ };
class Implementation { /* ... */ };

// OK: only one public base class, others may be non-public
class B: public Interface, private Implementation
{
    /* ... */
};
```

### 5.57.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each class definition, inspect the list of inheritances. If more than one base class is listed as public, emit a diagnostic.

### 5.57.5 References

This checker is a small part of the excellent discussion in: S. Meyers: "Effective C++ Second Edition", Item 43: "Use multiple inheritance judiciously".

## 5.58 Name All Parameters

This checker warn for anonymous parameters in function declarations and definitions. For definitions, if the argument is not used, a `static_cast<void>` should be used instead of not naming the parameter.

This checker check for the rule 22 from *The Elements of C++ Style* (Misfeldt and al., 2004).

### 5.58.1 Parameter Requirements

There is no parameter requirement.

### 5.58.2 Non-Compliant Code Example

```
void f(int)
{
}
```

### 5.58.3 Compliant Solution

To avoid warning messages from the compiler about unused variables, you can use a `static_cast<void>` to mark unused parameters.

```
void f(int i)
{
  static_cast<void>(i);
}
```

### 5.58.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For all function declarations, check that all parameter has a name.

### 5.58.5 References

Bumgardner G., Gray A., and Misfeldt T. *The Elements of C++ Style*. Cambridge University Press 2004.

## 5.59 [No Reference] : New Delete

This analysis checks for the validity of all delete operations in a specific source code. It checks for violations of:

a) Deleting an array with a simple delete operator instead of an delete array operator.

b) Deleting a NULL pointer.

c) Checks for the deletion of uninitialized pointers.

### 5.59.1 Non-Compliant Code Examples

In the following examples, a) b) and c) from above are demonstrated.

```
class Y {
  int y;
};

void fail() {
  int x=2;
  // deleting array
  Y* m = new Y[5];
  delete m;

  // deleting NULL
  Y* n = 0;
  delete n;

  // deleting NULL
  Y* c ;
  if (x==5) {
    x=7;
    delete c;
  }
}
```

### 5.59.2 Compliant Solution

Use assert statements to make sure that a pointer cannot be null, or use the delete[] operator if a new[] operator precedes on that pointer.

### 5.59.3 Parameter Requirements

None.

### 5.59.4 Implementation

This analysis uses the BOOST library in order to utilize the breadth first search (BFS) algorithm. Together with the control flow graph and the BFS, the implementation backtracks the code from the delete operation to its definition. On violations of the described cases, the analysis results in warnings.

The algorithm searches first for each occurance of a SgDeleteExp and backtracks this Node to its definition. If we find a SgNew operation, we need to see if the delete and new operations match, i.e. whether they are both operations on pointers or arrays.

The following cases are checked for and handled during the backwards dataflow analysis:

```
case V_SgNewExp:
case V_SgVarRefExp:
case V_SgAddressOfOp:
case V_SgCastExp:
case V_SgIntVal:
```

The above indicates a recursive algorithm.

### 5.59.5 References

## 5.60 No Asm Stmts Ops

### 5.60.1 Parameter Requirements

This checker takes no parameters and inputs source file

### 5.60.2 Implementation

This checker uses a simple AST traversal that checks for SgAsmStmt(s) and SgAsmOp(s). Any such nodes that are found are flagged as violations

### 5.60.3 Non-Compliant Code Example

This example is taken from Cxx_tests/test2006_98.C

```
typedef int _Atomic_word;

#ifndef __INTEL_COMPILER
// Intel complains that the input register "m" cannot have a modifier "+"
static inline _Atomic_word
__attribute__ ((__unused__))
__exchange_and_add (volatile _Atomic_word *__mem, int __val)
{
  register _Atomic_word __result;
  __asm__ __volatile__ ("lock; xadd{l} {%0,%1|%1,%0}"
                        : "=r" (__result), "+m" (*__mem)
                        : "0" (__val)
                        : "memory");
  return __result;
}
#endif
```

### 5.60.4 Compliant Solution

The compliant solution does not make use of C assembly.

### 5.60.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal visiting SgAsmStmt and SgAsmOp nodes.
2. For each such node flag violation.
3. Report all violations.

### 5.60.6 References

## 5.61 No Exceptions

This checker detects all usages of C++ exception handling.

### 5.61.1 Parameter Requirements

No parameters are required.

### 5.61.2 Implementation

The checker detects try statements, throw operations and catch statements.

### 5.61.3 Non-Compliant Code Example

```
class Exception{};

int main(){
 try {
    throw Exception();
 } catch( Exception e )
   { }
};
```

### 5.61.4 Compliant Solution

```
int main(){
};
```

### 5.61.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Traverse the AST
2. For each try statements, throw operations and catch statements report and error.

### 5.61.6 References

The ALE3D style guide section 17.1 forbids usage of C++ exceptions.

## 5.62 No Exit In Mpi Code

"ALE3D Coding Standards & Style Guide" item #19.1 states that

> `exit()` must never be called from a parallel code. Calling `exit()` from a parallel code will cause the code to deadlock. Even if you can guarantee that every processor will call `exit()` collectively, this can leave some parallel environments in a hung state because MPI resources are not properly cleaned up.

### 5.62.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.62.2 Implementation

This pattern is checked using a simple AST traversal seeking function reference expressions. These function reference expressions matching a call to the exit() function between blocks of MPI code (as delimited between MPI Init and MPI Finalize) are flagged as checker violations.

### 5.62.3 Non-Compliant Code Example

This trivial non-compliant code calls `exit()` from an MPI block.

```
#include <stdlib.h>
#include "mpi.h"

int main( int argc, char **argv )
{
  MPI_Init( &argc, &argv );
    exit(1);
  MPI_Finalize();

  return 0;
} //main()
```

### 5.62.4 Compliant Solution

The compliant solution uses `MPI_Abort()` instead.

```
#include <stdlib.h>
#include "mpi.h"

int main( int argc, char **argv )
{
  MPI_Init( &argc, &argv );
    MPI_Abort( MPI_COMM_WORLD, 1 );
  MPI_Finalize();
```

```
  return 0;
} //main()
```

### 5.62.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform a simple AST traversal of nodes that occur between `MPI_Init()` and `MPI_Finalize()` blocks.
2. For each node between MPI blocks, if node is call to `exit()` then flag violation.
3. Report any violations.

### 5.62.6 References

Arrighi B., Neely R., Reus J. "ALE3D Coding Standards & Style Guide", 2005.

## 5.63 No Goto

This checker detects uses of `goto` statements conforming with *High-Integrity C++ Coding Standard Manual*, rule 5.8: "Do not use `goto`"

### 5.63.1 Parameter Requirements

No parameter is needed

### 5.63.2 Non-Compliant Code Example

```
void foo() {
  tryAgain:
  try {
    doSomething();
  }
  catch (...) {
    goto tryAgain;
  }
}
```

### 5.63.3 Compliant Solution

```
void foo() {
  do {
    try {
      doSomething();
    }
    catch (...) {
      continue ;
    }
    break ;
  } while (true);
}
```

### 5.63.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Look for goto expressions.

### 5.63.5 References

The Programming Research Group, *High-Integrity C++ Coding Standard Manual*, rule 5.8: "Do not use `goto`".

## 5.64 Non Associative Relational Operators

*C++ Secure Coding Practices* states that:

> The relational and equality operators are left-associative, not non-associative as they often are in other languages. This allows a C++ programmer to write an expression (particularly an expression used as a condition) that can be easily misinterpreted.

This checker checks that relational binary operators ($==, != , <, >, <=, >=$) are not treated as if they were non-associative.

### 5.64.1 Parameter Requirements

This checker takes no parameters and inputs source file

### 5.64.2 Implementation

This pattern is checked using a nested AST traversal on the parent nodes of the operand of a binary operator expression. Any such parent node that treats relational binary operators as non-associative will use more than one binary relational operator. Flag these expressions as violations.

### 5.64.3 Non-Compliant Code Example

```
#include <stdio.h>

int main()
{
  int a = 2;
  int b = 2;
  int c = 2;

  if ( a < b < c ) // condition #1, misleading, likely bug
    printf( "a < b < c\n" );
  if ( a == b == c ) // condition #2, misleading, likely bug
    printf( "a == b == c\n" );

  return 0;
}
```

### 5.64.4 Compliant Solution

```
#include <stdio.h>

int main()
{
  int a = 2;
  int b = 2;
  int c = 2;
```

```
  if ( a < b && b < c ) // clearer, and probably what was intended
    printf( "a < b && b < c\n" );
  if ( a == b && a == c ) // ditto
    printf( "a == b && a == c\n" );

  return 0;
}
```

### 5.64.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal and locate SgBinaryOp nodes.
2. At each SgBinaryOp node perform a nested traversal of the operands parent node and count the number of relational binary operators used.
3. If said count is greater than one then flag error.
4. Report all errors.

### 5.64.6 References

EXP09-A. Treat relational and equality operators as if they were nonassociative

## 5.65 Nonmember Function Interface Namespace

User-defined classes should typically reside in the same namespace as their nonmember function interface, i.e. friend functions and operators for that class. The reasons are uniform lookup of overloaded functions and that the interface of a class consists not only of its member functions. This checker enforces this guideline. It reports friend function declarations that refer to functions from a different namespace. Further, it makes sure that every class mentioned in a nonmember operator's signature (return and argument types) is in the same namespace as the operator, or in the global std namespace.

### 5.65.1 Parameter Requirements

No parameters are required.

### 5.65.2 Non-Compliant Code Example

```
void f(); // not OK: used as friend in class N::A, not in same namespace
namespace N
{
    class A
    {
    public:
        friend void ::f();
    };
}
```

### 5.65.3 Compliant Solution

```
namespace M
{
    void f(); // OK: used as friend in M::B, same namespace

    class B
    {
    public:
        friend void f();
    };
}
```

### 5.65.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each friend function or operator for a class, look up the namespace it is declared in and compare to the namespace of the class.

### 5.65.5 References

The reference for this checker is: H. Sutter, A. Alexandrescu: "C++ Coding Standards", Item 57: "Keep a type and its nonmenber function interface in the same namespace".

## 5.66 Non Standard Type Ref Args

Per the Abbreviated C++ Code Inspection Checklist "While it is cheaper to pass ints, longs, and such by value, passing objects this way incurs significant expense due to the construction of temporary objects. The problem becomes more severe when inheritance is involved. Simulate pass-by-value by passing const references."

### 5.66.1 Parameter Requirements

No parameters necessary

### 5.66.2 Implementation

The arguments to all functions are checked for base type in the declaration. If the base type is found to be a struct or a class, it is then checked to ensure it is a reference. If it is not, a notification is raised.

### 5.66.3 Non-Compliant Code Example

```
class incrediblyComplex
{
private:
//loads of members
}

bool justLooking(incrediblyComplex fullCopy)
{
return (! &fullCopy);
}
```

### 5.66.4 Compliant Solution

```
class incrediblyComplex
{
private:
//loads of members
}

bool justLooking(incrediblyComplex& fullCopy)
{
return (! &fullCopy);
}
```

### 5.66.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Identify function declaration
2. Check arguments for base type
3. if non-intrinsic type and not a reference, notify

### 5.66.6 References

Abbreviated Code Inspection Checklist Section 13.1, Argument Passing"

## 5.67 Non Standard Type Ref Returns

While it is cheaper to pass ints, longs, and such by value, passing objects this way incurs significant expense due to the construction of temporary objects. The problem becomes more severe when inheritance is involved. Simulate pass-by-value by passing const references.

### 5.67.1 Parameter Requirements

No parameters necessary.

### 5.67.2 Implementation

The return types to all functions are checked for base type in the declaration. If the base type is found to be a struct or a class, it is then checked to ensure it is a reference. If it is not, a notification is raised.

### 5.67.3 Non-Compliant Code Example

```
class incrediblyComplex
{
private:
//loads of members
}

incrediblyComplex justLooking()
{
incrediblyComplex *fullCopy = new incrediblyComplex();
return (*fullCopy);
}
```

### 5.67.4 Compliant Solution

```
class incrediblyComplex
{
private:
//loads of members
}

incrediblyComplex& justLooking()
{
incrediblyComplex *fullCopy = new incrediblyComplex();
return (*fullCopy);
}
```

### 5.67.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Identify function declaration
2. Check return for base type
3. if non-intrinsic type and not a reference, notify

### 5.67.6 References

Abbreviated Code Inspection Checklist Section 14.5, Return Values"

## 5.68 Non Virtual Redefinition

Calls of nonvirtual member functions are resolved at compile time, not run time. Redefinition of an inherited nonvirtual function in a derived class has different semantics that can result in surprising behavior and should therefore be avoided. This checker reports cases where a class redefines a function that was declared nonvirtual in one of its superclasses.

### 5.68.1 Parameter Requirements

This checker does not require any parameters.

### 5.68.2 Non-Compliant Code Example

```
namespace NonCompliant {
    class Base {
    public:
        virtual void overrideIfYouWish(int);
        void doNotOverride(int);
    };

    class Inherited: public Base {
    public:
        void doNotOverride(int); // trying to override nonvirtual function
    };
}
```

### 5.68.3 Compliant Solution

```
namespace Compliant {
    class Base {
    public:
        virtual void overrideIfYouWish(int);
        void doNotOverride(int);
    };

    class Inherited: public Base {
    public:
        void overrideIfYouWish(int); // overriding virtual function
    };
}
```

### 5.68.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For every member function declaration, traverse the inheritance DAG of the enclosing class.
2. Identify any functions that may be overridden by the current one (by checking name and type).
3. Issue a diagnostic if the overridden function is not declared virtual.

### 5.68.5 References

A reference for this checker is: S. Meyers: "Effective C++ Second Edition", Item 37: "Never redefine an inherited nonvirtual function."

## 5.69 No Overload Ampersand

The C++ standard [ISO/IEC 14882-2003] says in Section 5.3.1 paragraph 4 that

The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares operator&() as a member function, then the behavior is undefined (and no diagnostic is required).

Therefore, to avoid possible undefined behavior, the operator & should not be overloaded.

### 5.69.1 Parameter Requirements

No Parameters Required.

### 5.69.2 Implementation

We check any member function then compare the name to 'operator&'. If this combination is found an alert is raised.

### 5.69.3 Non-Compliant Code Example

```
class peanutButter
{
  string name;
  void operator&()
  {
    name += '&jelly';
  }
}
```

### 5.69.4 Compliant Solution

```
class peanutButter
{
  string name;
  void addJelly()
  {
    name += '&jelly';
  }
}
```

### 5.69.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Find member function
2. Check name
3. raise alert

### 5.69.6 References

ISO/IEC 9899-1999:TC2 ISO/IEC 14882-2003 Section 5.3.1, “Unary operators” Lockheed Martin 05 AV Rule 159, “Operators ||, &&, and unary & shall not be overloaded”

## 5.70 CERT-MSC30-C: No Rand

"CERT Secure Coding MSC30-C" states

> The C Standard function `rand` (available in `stdlib.h`) does not have good random number properties. The numbers generated by `rand` have a comparatively short cycle, and the numbers may be predictable. To achieve the best random numbers possible, an implementation-specific function needs to be used.

### 5.70.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.70.2 Implementation

This pattern is checked using a simple AST traversal that visits all function reference expressions. If a functino reference expression node corresponds to the `rand()` function, then a violation is flagged.

### 5.70.3 Non-Compliant Code Example

The following code calls `rand()`.

```
1 #include <stdlib.h>
2
3 int main()
4 {
5   int r = rand();  /* generate a random integer */
6
7   return 0;
8 }
```

### 5.70.4 Compliant Solution

The compliant solution is to use a implementation-specific random number generator.

```
1 int main()
2 {
3   int r = my_rand();  /* generate a random integer */
4
5   return 0;
6 }
```

### 5.70.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform a simple AST traversal visiting all function reference expression nodes.

2. For each node visited, if the function reference expression corresponds to `rand()` then flag violation.

3. Report any violations.

### 5.70.6 References

Secure Coding : MSC30-C. Do not use the rand function

## 5.71 No Second Term Side Effects

The logical AND and logical OR operators (&&, ||) exhibit "short circuit" operation. That is, the second operand is not evaluated if the result can be deduced solely by evaluating the first operand. Consequently, the second operand should not contain side effects because, if it does, it is not apparent if the side effect occurs

### 5.71.1 Parameter Requirements

No parameters required.

### 5.71.2 Implementation

We check for And or Or. We then query for any of a set of operators known to have side effects. This checker has the known deficiency of not checking function calls for side-effects. To avoid false positives, it does not notify of functions at all.

### 5.71.3 Non-Compliant Code Example

```
int i;
int max;

if ( (i >= 0 && (i++) <= max) ) {
  /* code */
}
```

It is unclear whether the value of i will be incremented as a result of evaluating the condition.

### 5.71.4 Compliant Solution

In this compliant solution, the behavior is much clearer.

```
int i;
int max;

if ( (i >= 0 && (i + 1) <= max) ) {
  i++;
  /* code */
}
```

### 5.71.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Find the And or Or operator

2. query the right-hand child for known side-effect having operators

### 5.71.6 References

ISO/IEC 9899-1999:TC2 [ISO/IEC 9899-1999] Section 6.5.13, “Logical AND operator,” and Section 6.5.14, “Logical OR operator”

## 5.72 Secure Coding : EXP06-A. Operands to the sizeof operator should not contain side effects

The `sizeof` operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. If the type of the operand is not a variable length array type the operand is **not** evaluated.

Providing an expression that appears to produce side effects may be misleading to programmers who are not aware that these expressions are not evaluated. As a result, programmers may make invalid assumptions about program state leading to errors and possible software vulnerabilities.

### 5.72.1 Non-Compliant Code Example

In this example, the variable `a` will still have a value 14 after `b` has been initialized.

```
1 int a = 14;
2 int b = sizeof(a++);
3
```

The expression `a++` is not evaluated. Consequently, side effects in the expression are not executed.

#### Implementation Specific Details

This example compiles cleanly under Microsoft Visual Studio 2005 Version 8.0, with the /W4 option.

### 5.72.2 Compliant Solution

In this compliant solution, the variable `a` is incremented.

```
1 int a = 14;
2 int b = sizeof(a);
3 a++;
4
```

#### Implementation Specific Details

This example compiles cleanly under Microsoft Visual Studio 2005 Version 8.0, with the /W4 option.

### 5.72.3 Risk Assessment

If expressions that appear to produce side effects are supplied to the `sizeof` operator, the returned result may be different then expected. Depending on how this result is used, this could lead to unintended program behavior.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---|---|---|---|---|---|
| EXP06-A | **1** (low) | **1** (unlikely) | **3** (low) | **P3** | **L3** |

**Related Vulnerabilities**

Search for vulnerabilities resulting from the violation of this rule on the CERT website .

### 5.72.4 References

[ ISO/IEC 9899-1999 ] Section 6.5.3.4, "The sizeof operator"

## 5.73 No Template Usage

Finds all usages of C++ templates. It will not detect C++ template declarations that are not instantiated.

### 5.73.1 Parameter Requirements

No parameters are required.

### 5.73.2 Implementation

The checker finds all template instatiation declaration, template instatiation definitions, template instantiation member function declarations and template instatiation function declarations.

### 5.73.3 Non-Compliant Code Example

```
template<typename t>
class Foo
{
   public:
      Foo(){};
      ~Foo(){};
};

void main()
{
 Foo<int>    fi;
 Foo<float> ff;
}
```

### 5.73.4 Compliant Solution

```
class Foo
{
   public:
      Foo(){};
      ~Foo(){};
};

void main()
{
 Foo    fi;
 Foo    ff;
}
```

### 5.73.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Traverse the AST
2. For each template instantiations and template declaration report an error

### 5.73.6 References

The ALE3D style guide section 16.1 states that templates must not be used.

## 5.74 No Variadic Functions

"CERT Secure Coding DCL33-C." states

> A variadic function  a function declared with a parameter list ending with ellipsis (...)  can accept a varying number of arguments of differing types. Variadic functions are flexible, but they are also hazardous. The compiler can't verify that a given call to a variadic function passes an appropriate number of arguments or that those arguments have appropriate types. Consequently, a runtime call to a variadic function that passes inappropriate arguments yields undefined behavior. Such undefined behavior could be exploited to run arbitrary code.

### 5.74.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.74.2 Implementation

This pattern is checked using a simple AST traversal that visits all function and member function references checking the function declaration for arguments of variadic type. Those defined functions with variadic arguments flag violations of this rule.

### 5.74.3 Non-Compliant Code Example

```
#include <cstdarg>

char *concatenate(char const *s, ...)
{
  return 0;
}

int main()
{
  char *separator = "\t";
  char *t = concatenate("hello", separator, "world", 0);

  return 0;
}
```

### 5.74.4 Compliant Solution

The compliant solution uses a chain of string binary operations instead of a variadic function.

```
#include <string>

string separator = /* some reasonable value */;
```

```
string s = "hello" + separator + "world";
```

### 5.74.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal on all function and member function references.
2. For each function reference check the function declaration and existence of function definition.
3. If function definition does not exist then stop check.
4. Else check function declaration arguments for variadic types.
5. Report any violations.

### 5.74.6 References

DCL33-C. Do not define variadic functions

## 5.75 CERT-POS33-C: No Vfork

"CERT Secure Coding POS33-C" states

> Using the `vfork` function introduces many portability and security issues. There are many cases in which undefined and implementation specific behavior can occur, leading to a denial of service vulnerability.

### 5.75.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.75.2 Implementation

This pattern is checked using a simple AST traversal that visits all function reference expressions. If a functino reference expression node corresponds to the `vfork()` function, then a violation is flagged.

### 5.75.3 Non-Compliant Code Example

This non-compliant example calls `vfork()`.

```
1 #include <stdlib.h>
2 #include <unistd.h>
3
4 int main()
5 {
6   pid_t pid = vfork();
7
8   if ( pid == 0 )  /* child */
9   {
10    system( "echo \"Hello World\"" );
11  }
12
13  return 0;
14 }
```

### 5.75.4 Compliant Solution

The compliant solution calls `fork()` instead.

```
1 #include <stdlib.h>
2 #include <unistd.h>
3
4 int main()
5 {
6   pid_t pid = fork();
7
8   if ( pid == 0 )  /* child */
9   {
10    system( "echo \"Hello World\"" );
11  }
12
13  return 0;
14 }
```

### 5.75.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform a simple AST traversal visiting all function reference expressions.
2. For each node visited, if the function reference expression corresponds to `vfork()` then flag violation.
3. Report any violations.

### 5.75.6 References

Secure Coding : POS33-C. Do not use vfork()

## 5.76 CERT EXP33-C and EXP34-C : Null Dereference

NULL Dereference checker. If any variable that could be NULL is dereferenced, a warning is issued. This is an implementation of US-CERT rules: EXP33-C - Do not reference uninitialized variables and EXP34-C - Ensure a pointer is valid before dereferencing it.

**EXP33-C - Do not reference uninitialized variables**

Local, automatic variables can assume unexpected values if they are used before they are initialized. C99 specifies *If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate* [ISO/IEC 9899-1999]. In practice, this value defaults to whichever values are currently stored in stack memory. While uninitialized memory often contains zero, this is not guaranteed. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.

### 5.76.1 EXP34-C - Ensure a pointer is valid before dereferencing it

Attempting to dereference an invalid pointer results in undefined behavior, typically abnormal program termination. Given this, pointers should be checked to make sure they are valid before they are dereferenced.

### 5.76.2 Non-Compliant Code Examples

**EXP33-C - Do not reference uninitialized variables**

In this example, the set_flag() function is supposed to set a the variable sign to 1 if number is positive and -1 if number is negative. However, the programmer forgot to account for number being 0. If number is 0, then sign will remain uninitialized. Because sign is uninitialized, it assumes whatever value is at that location in the program stack. This may lead to unexpected, incorrect program behavior.

```
void set_flag(int number, int *sign_flag) {
  if (number > 0) {
    *sign_flag = 1;
  }
  else if (number <0) {
    *sign_flag = -1;
  }
  int x = *sign_flag;
}

int main(int argc, char** argv) {
  int sign;
  set_flag(0,&sign);
```

```
  return 0;
}
```

**EXP34-C - Ensure a pointer is valid before dereferencing it**

In this example, input_str is copied into dynamically allocated memory referenced by str. If malloc() fails, it returns a NULL pointer that is assigned to str. When str is dereferenced in strcpy(), the program behaves in an unpredictable manner.

```
#include "assert.h"
#include <stdlib.h>

void testme() {
  // case 1
  int size = 5;
  char* str = (char*) malloc(size+1);
  char z = *str;

  // case 2
  int *p = 0;
  int l = *p;

  // case 3
  char *k=0;
  free(k);

}
```

### 5.76.3 Compliant Solution

**EXP33-C - Do not reference uninitialized variables**

We do not check the expressions in if conditions, and hence it is irrelevant what the if conditions state. However, because an if condition occurs, there might be a path that leaves sign_flag uninitialized. In this case a simple assert helps to avoid the warning caused by this analysis.

```
#include "assert.h"

void set_flag(int number, int *sign_flag) {
  assert(sign_flag);
  if (number > 0) {
    *sign_flag = 1;
  }
  else if (number <0) {
    *sign_flag = -1;
  }
  int x = *sign_flag;
}
```

```
int main(int argc, char** argv) {
  int sign;
  set_flag(0,&sign);
  return 0;
}
```

### 5.76.4 EXP34-C - Ensure a pointer is valid before dereferencing it

```
#include "assert.h"
#include <stdlib.h>

void testme() {
  // case 1
  int size = 5;
  char* str = (char*) malloc(size+1);
  if (str==NULL) {
    *str = '5';
  }
  char z = *str;

  // case 2
  int *p = 0;
  assert(p);
  int l = *p;

  // case 3
  char *k=0;
  assert(k);
  free(k);

}
```

### 5.76.5 Parameter Requirements

None.

### 5.76.6 Implementation

We use a dataflow analysis to determine null dereference. The dataflow analysis is based on an *breadth first search* (bfs) algorithm, implemented in BOOST. The implementation does a bfs backwards traversal for each:

a) SgArrowExp
b) SgPointerDerefExp
c) SgAssignInitializer
d) SgFunctionCallExp (free)

The above are the points that need to be validated by the algorithm. At each such point the program might be invalid due to NULL pointer dereferences.

Therefore, the variable at that point must be determined and the programmed is tracked back (dataflow).

If at any point an assertion is found, the analysis is aborted for that run, i.e. no null pointer dereference is present.

### 5.76.7 References

EXP33C , "Do not reference uninitialized variables"
EXP34C , "Ensure a pointer is valid before dereferencing it"

## 5.77 CERT-DCL04-A: One Line Per Declaration

"CERT Secure Coding DCL04-A" states

> Declaring multiple variables on a single line of code can cause confusion regarding the types of the variables and their initial values. If more than one variable is declared on a line, care must be taken that the actual type and initialized value of the variable is known. To avoid confusion, more than one type of variable should not be declared on the same line.

### 5.77.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.77.2 Implementation

This pattern is checked using a simple AST traversal, visiting all variable declaration statements. The line number of each variable declaration statement node is saved to a std set unique to each file. If any line number is added to this set more than once then a violation is flagged.

### 5.77.3 Non-Compliant Code Example

The non-compliant code declares multiple `int` variables on the same line.

```
int main()
{
  int i1 = 0, i2 = 0, i3 = 0;

  return 0;
}
```

### 5.77.4 Compliant Solution

The compliant solution is to give each `int` declaration its own line.

```
int main()
{
  int i1 = 0;
  int i2 = 0;
  int i3 = 0;

  return 0;
}
```

### 5.77.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform a simple AST traversal visiting all variable declaration nodes.
2. For each line number associated with a variable declaration node, add the line number to a set of line numbers unique to its source file.
3. If any line number is added more than once per source file set of line numbers then flag a violation.
4. Report any violations.

### 5.77.6 References

Secure Coding : DCL04-A. Take care when declaring more than one variable per line

## 5.78 Operator Overloading

This test detects function declaration that overloads operators that can cause subtle bugs, such as "`&&`", "`||`", or "`,`". That is, one can not ensure that the overloaded operators will be evaluated in left-to-right order. This is based on the following rules:

- Function calls always evaluate all arguments before execution.
- The order of evaluation of function arguments is unspecified.

For example,

```
auto_ptr<Employee> e = GetEmployee();
if(e && e->Manager())
```

The usual evaluation order (left to right) prevents the test from executing `e->Manager()` and the code above looks fine. However, the code above can invoke an overloaded `operator&&` and it will potentially call `e->Manager()` before checking if `e` is `NULL`.

### 5.78.1 Parameter Requirements

None.

### 5.78.2 Implementation

This pattern is detected using a simple traversal. It traverses AST to find function declarations and check whether or not the name of the functions is "`operator&&`", "`operator||`", or "`operator,`".

### 5.78.3 Non-Compliant Code Example

```
class Test
{
  public:
    Test();
    ~Test();
    Test operator&&(const Test &);
    Test operator||(const Test &);
    Test operator,(const Test &);
};
```

### 5.78.4 Compliant Solution

```
N/A
```

### 5.78.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Check if a node is a function declaration
2. Check if the name of the function contains "`operator&&`", "`operator||`", or "`operator,`".

### 5.78.6 References

T. Misfeldt, G. Bumgardner, A. Gray, "The Elements of C++ Style", Item 111: "Do not overload `operator &&` or `operator ||`".

## 5.79 Other Argument

This checker enforce the name convention of the first argument in copy constructors and copy operators. This is taken from rule 23 from *the Elements of C++ Style* (Misfeldt and al., 2004). The parameter should be called `other`. This checker also accept two other naming conventions: `that` and the class name in lower camel case.

### 5.79.1 Parameter Requirements

There is no parameter requirement.

### 5.79.2 Non-Compliant Code Example

```
A::A(const A& foo)
{
  //...
}
```

### 5.79.3 Compliant Solution

```
A::A(const A& other)
{
  //...
}
```

### 5.79.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For all constructors and operator fulfilling the copy requirement of the C++ standard, check that the first parameter is of the three possible name.

### 5.79.5 References

Bumgardner G., Gray A., and Misfeldt T. *The Elements of C++ Style*. Cambridge University Press 2004.

## 5.80 Place Constant On The Lhs

This checker detects a test clause whether or not it contains a constant on the left hand side when comparing a varialbe and the constant for equality. By putting the constant on the left hand side, the compiler can prevent programmers from making mistake to write '=' for '=='.

### 5.80.1 Parameter Requirements

None

### 5.80.2 Implementation

This checker is implemented with a simple traversal. It traverses AST and finds a test clause. If the test clause has a variable on its left hand side, then, then the checker report this clause to the standard output.

### 5.80.3 Non-Compliant Code Example

```
void foo()
{
int a = 0;

if(a == 10) // a is on the LHS
{
a = 1;
}

while(a == 10) // a is on the LHS
{
a++;
}

do
{
a++;
}while(a == 12); // a is on the LHS

for(int i = 0; i == 0; i++) // i is on the LHS
{
a = 12;
}
}
```

### 5.80.4 Compliant Solution

```
void foo()
{
int a = 0;
```

```
if(1 == a) // fine
{
a = 2;
}
}
```

### 5.80.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Check if the node visiting is an if statement.
2. If yes, find its test clause to see if it contains a constant on the left hand side.
3. Check also if the if statement compares a variable and the constant for equality

### 5.80.6 References

The Programming Research Group: High-Integrity C++ Coding Standard Manual, Item 10.6: "When comparing variables and constants for equality always place place the constnat on the left hand side."

## 5.81 Prefer Algorithms

Many people consider hand-written loops over STL containers inferior to calls to STL algorithms for reasons of efficiency, correctness, and maintainability.

This checker is meant to highlight cases where a loop might be replaced by an equivalent STL algorithm call. It reports for loops where the loop head fulfills the following properties:

- The initialization part contains an assignment or variable declaration with an initializer,
- the condition part consists of an inequality comparison, and
- the increment part consists of an increment or decrement operation.

For loops on integer or floating-point types are not reported as those cannot be replaced by STL algorithms.

### 5.81.1 Parameter Requirements

This checker does not require any parameters.

### 5.81.2 Non-Compliant Code Example

```
#include <vector>

void add_x_to_each_element_noncompliant(int x, std::vector<int> &v)
{
    // not OK: loop to add x to each element
    std::vector<int>::iterator v_itr;
    for (v_itr = v.begin(); v_itr != v.end(); ++v_itr)
        *v_itr += x;
}
```

### 5.81.3 Compliant Solution

```
#include <vector>
#include <algorithm>
#include <functional>

void add_x_to_each_element_compliant(int x, std::vector<int> &v)
{
    // OK: using an algorithm to add x to each element
    transform(v.begin(), v.end(), v.begin(),
              std::bind2nd(std::plus<int>(), x));
}
```

### 5.81.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each for loop, check the criteria explained above, taking both built-in and overloaded operators.
2. If the loop fulfills all criteria, generate a diagnostic.

### 5.81.5 References

A reference for this checker is: S. Meyers: "Effective STL", Item 43: "Prefer algorithm calls to hand-written loops".

## 5.82 Secure Coding : FIO07-A. Prefer fseek() to rewind()

`rewind()` sets the file position indicator for a stream to the beginning of that stream. However, `rewind()` is equivalent to `fseek()` with `0L` for the offset and `SEEK_SET` for the mode with the error return value suppressed. Therefore, to validate that moving back to the beginning of a stream actually succeeded, `fseek()` should be used instead of `rewind()`.

### 5.82.1 Non-Compliant Code Example

The following non-compliant code sets the file position indicator of an input stream back to the beginning using `rewind()`.

```
1 FILE* fptr = fopen("file.ext", "r");
2 if (fptr == NULL) {
3   /* handle open error */
4 }
5
6 /* read data */
7
8 rewind(fptr);
9
10 /* continue */
11
```

However, there is no way of knowing if `rewind()` succeeded or not.

### 5.82.2 Compliant Solution

This compliant solution instead using `fseek()` and checks to see if the operation actually succeeded.

```
1 FILE* fptr = fopen("file.ext", "r");
2 if (fptr == NULL) {
3   /* handle open error */
4 }
5
6 /* read data */
7
8 if (fseek(fptr, 0L, SEEK_SET) != 0) {
9   /* handle repositioning error */
10 }
11
12 /* continue */
13
```

### 5.82.3 Risk Assessment

Using `rewind()` makes it impossible to know whether the file position indicator was actually set back to the beginning of the file. If the call does fail, the result may be incorrect program flow.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---|---|---|---|---|---|
| FIO07-A | **1** (low) | **1** (unlikely) | **2** (medium) | **P2** | **L3** |

**Related Vulnerabilities**

Search for vulnerabilities resulting from the violation of this rule on the CERT website .

### 5.82.4 References

[ ISO/IEC 9899-1999:TC2 ] Section 7.19.9.2, "The `fseek` function"; 7.19.9.5, "The `rewind` function"

## 5.83 Prefer Setvbuf To Setbuf

The functions setvbuf() and setbuf() are defined as follows:

void setbuf(FILE * restrict stream, char * restrict buf); int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);

setvbuf() is equivalent to setbuf() with _IOFBF for mode and BUFSIZE for size (if buf is not NULL) or _IONBF for mode (if buf is NULL), except that it returns a nonzero value if the request could not be honored. For added error checking, prefer using setvbuf() over setbuf().

### 5.83.1 Parameter Requirements

No Parameter specifications.

### 5.83.2 Implementation

No implementation yet!

### 5.83.3 Non-Compliant Code Example

The following non-compliant code makes a call to setbuf() with an argument of NULL to ensure an optimal buffer size.

```
% write your non-compliant code example
FILE* file;
char *buf = NULL;
/* Setup file */
setbuf(file, buf);
/* ... */
```

However, there is no way of knowing whether the operation succeeded or not.

### 5.83.4 Compliant Solution

This compliant solution instead calls setvbuf(), which returns nonzero if the operation failed.

```
% write your compliant code example
FILE* file;
char *buf = NULL;
/* Setup file */
if (setvbuf(file, buf, buf ? _IOFBF : _IONBF, BUFSIZ) != 0) {
  /* Handle error */
}
/* ... */
```

### 5.83.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Write your checker algorithm

### 5.83.6 References

ISO/IEC9899-1999:TC2 FIO12-A. Prefer setvbuf() to setbuf()

## 5.84 Protect Virtual Methods

*The Elements of C++ Style* item #107 states

> Do not expose virtual methods in the public interface of a class. Use a public methods with a similar name to call the protected virtual method.

### 5.84.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.84.2 Implementation

This pattern is checked using a simple AST traversal that seeks instances of SgMemberFunctionDeclaration that have the public access modifier and the virtual function modifier boolean values set to true. Member functions that match this pattern are flagged as violations.

### 5.84.3 Non-Compliant Code Example

This non-compliant code contains a virtual function in the public interface of a class.

```
class Class
{
  int n;

  public:
    Class() { n = publicVirtualFunction(); } //constructor
    ~Class() {} //Destructor

    virtual int publicVirtualFunction() { return 1; }
}; //class Class
```

### 5.84.4 Compliant Solution

The compliant solution protects the virtual function and adds a public accessor to the virtual function.

```
class Class
{
  int n;

  protected:
    virtual int protectedVirtualFunction() { return 1; }
  public:
    Class() { n = publicVirtualFunction(); } //constructor
    ~Class() {} //Destructor
    int publicVirtualFunction(){ return protectedVirtualFunction(); }
}; //class Class
```

### 5.84.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform a simple AST traversal visiting all function declaration nodes.
2. For each function declaration check the "public" and "virtual" modifiers. If both "public" and "virtual" modifiers are set then flag violation.
3. Report any violations.

### 5.84.6 References

Bumgardner G., Gray A., and Misfeldt T. *The Elements of C++ Style*. Cambridge University Press 2004.

## 5.85 Push Back

Tests if the source uses front or back insertion in sequences using insert or resize where a `push_front` or `push_back` could be used. The patterns are very simple and matches simple calls like `vector.insert(vector.end(), ...)`.

In these case, `push_back` and `push_front` only are insured to be efficient. All other calls may be quadratic.

This test is inspired by the rule 80 of C++ Coding Standards: "Use the accepted idioms to really shrink capacity and really erase elements".

### 5.85.1 Parameter Requirements

There is no parameter required.

### 5.85.2 Implementation

No implementation yet!

### 5.85.3 Non-Compliant Code Example

```
#include <vector>
#include <list>

void g()
{
  std::vector<int> v;
  v.insert(v.end(), 1);

  v.resize(v.size() + 1, 1);

  std::list<int>* vv = new std::list<int>();
  vv->insert(vv->begin(), 1);

}
```

### 5.85.4 Compliant Solution

```
#include <vector>
#include <list>

void g()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(1);

  std::list<int>* vv = new std::list<int>();
  vv->push_front(1);
```

```
}
```

### 5.85.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers looking for these patterns:

For all types `T`,
where `V` variable of type `vector<T>`,
where `Vp` variable of type `vector<T>*`,
where `L` variable of type `list<T>`,
where `Lp` variable of type `list<T>*`,
where `S` variable of type `slist<T>`,
where `Sp` variable of type `slist<T>*`,
where `D` variable of type `deque<T>`,
where `Dp` variable of type `deque<T>*`,
where `Value` expression of type `T`,
the check will look for these patterns in the source code.

```
V.resize(V.size() + 1, Value)
Vp->resize(Vp->size() + 1, Value)
D.resize(D.size() + 1, Value)
Dp->resize(Dp->size() + 1, Value)
L.resize(L.size() + 1, Value)
Lp->resize(Lp->size() + 1, Value)
V.resize(1 + V.size(), Value)
Vp->resize(1 + Vp->size(), Value)
D.resize(1 + D.size(), Value)
Dp->resize(1 + Dp->size(), Value)
L.resize(1 + L.size(), Value)
Lp->resize(1 + Lp->size(), Value)
V.insert(V.end(), Value)
Vp->insert(Vp->end(), Value)
D.insert(D.end(), Value)
Dp->insert(Dp->end(), Value)
L.insert(L.end(), Value)
Lp->insert(Lp->end(), Value)
S.insert(S.begin(), Value)
Sp->insert(Sp->begin(), Value)
D.insert(D.begin(), Value)
Dp->insert(Dp->begin(), Value)
L.insert(L.begin(), Value)
Lp->insert(Lp->begin(), Value)
```

For all `resize` and back `insert` patterns, `push_back` could be used. For front `insert` patterns, `push_front` could be used instead.

### 5.85.6 References

Alexandrescu A. and Sutter H. *C++ Coding Standards 101 Rules, Guidelines, and Best Practices*. Addison-Wesley 2005.

## 5.86 Right Shift Mask

Do not assume that a right shift operation is implemented as either an arithmetic (signed) shift or a logical (unsigned) shift. If E1 in the expression E1 ¿¿ E2 has a signed type and a negative value, the resulting value is implementation defined and may be either an arithmetic shift or a logical shift. Also, be careful to avoid undefined behavior while performing a bitwise shift.

### 5.86.1 Parameter Requirements

No Parameter Required.

### 5.86.2 Implementation

Upon finding a right shift we trace parent pointers up until we find a bit and operator. If we find this bitwise and then we return. If we make it to the basic block node of the statement we raise an alert.

### 5.86.3 Non-Compliant Code Example

For implementations in which an arithmetic shift is performed and the sign bit can be propagated as the number is shifted.

```
int stringify;
char buf[sizeof("256")];
sprintf(buf, "%u", stringify >> 24);
% write your non-compliant code example
```

If stringify has the value 0x80000000, stringify ¿¿ 24 evaluates to 0xFFFFFF80 and the subsequent call to sprintf() results in a buffer overflow.

### 5.86.4 Compliant Solution

For bit extraction, make sure to mask off the bits you are not interested in.

```
int stringify;
char buf[sizeof("256")];
sprintf(buf, "%u", ((number >> 24) & 0xff));
% write your compliant code example
```

### 5.86.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Find the bitwise and
2. climb parent tree

3. alert if parent null or basicblock found.

### 5.86.6 References

ISO/IEC 9899-1999:TC2 [Dowd 06] Chapter 6, "C Language Issues" [ISO/IEC 9899-1999] Section 6.5.7, "Bitwise shift operators" [ISO/IEC 03] Section 6.5.7, "Bitwise shift operators"

## 5.87 Set Pointers To Null

Dangling pointers can lead to exploitable double-free and access-freed-memory vulnerabilities. A simple yet effective way to eliminate dangling pointers and avoid many memory related vulnerabilities is to set pointers to NULL after they have been freed. Calling free() on a NULL pointer results in no action being taken by free().

### 5.87.1 Parameter Requirements

No Parameter specifications.

### 5.87.2 Implementation

### 5.87.3 Non-Compliant Code Example

In this example, the type of a message is used to determine how to process the message itself. It is assumed that message_type is an integer and message is a pointer to an array of characters that were allocated dynamically. If message_type equals value_1, the message is processed accordingly. A similar operation occurs when message_type equals value_2. However, if message_type == value_1 evaluates to true and message_type == value_2 also evaluates to true, then message will be freed twice, resulting in an error.

```
if (message\_type == value\_1) {
  /* Process message type 1 */
  free(message);
}
/* ...*/
if (message\_type == value\_2) {
   /* Process message type 2 */
  free(message);
}
```

### 5.87.4 Compliant Solution

As stated above, calling free() on a NULL pointer results in no action being taken by free(). By setting message equal to NULL after it has been freed, the double-free vulnerability has been eliminated.

```
if (message_type == value_1) {
  /* Process message type 1 */
  free(message);
  message = NULL;
}
/* ...*/
if (message_type == value_2) {
```

```
  /* Process message type 2 */
  free(message);
  message = NULL;
}
```

### 5.87.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Write your checker algorithm

### 5.87.6 References

ISO/IEC 9899-1999 MEM01-A. Eliminate dangling pointers

## 5.88 Single Parameter Constructor Explicit Modifier

*The Elements of C++ Style* item #104 states that

> A compiler can use a single parameter constructor for type conversions. While this is natural in some situations, it might be unexpected in others... you can avoid this behavior by declaring a constructor with the `explicit` keyword.

### 5.88.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.88.2 Implementation

This pattern is checked using a simple AST traversal that finds instances of SgFunctionDeclaration that are constructors with a single parameter. If these SgFunctionDeclaration are not modified with the "explicit" keyword then a violation is flagged.

### 5.88.3 Non-Compliant Code Example

This non-compliant code has a single parameter constructor that is not declared with the `explicit` keyword.

```
class Class
{
  int num;
  public:
    Class( int n ){ num = n; }
    int getNum() const { return num; }
}; //class Class
```

### 5.88.4 Compliant Solution

The compliant solution declares the single parameter constructor with the `explicit` keyword modifier.

```
class Class
{
  int num;
  public:
    explicit Class( int n ){ num = n; }
    int getNum() const { return num; }
}; //class Class
```

### 5.88.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform a simple AST traversal visiting all function declaration nodes.
2. For each function declaration, if node is constructor then check the size of its parameter list.
3. If the parameter list size of constructor is 1 and is not declared with the `explicit` modifier then flag violation.
4. Report any violations.

### 5.88.6 References

Bumgardner G., Gray A., and Misfeldt T. *The Elements of C++ Style*. Cambridge University Press 2004.

## 5.89 Size Of Pointer

Do not take the size of a pointer to a type when you are trying to determine the size of the type. Taking the size of a pointer to a type always returns the size of the pointer and not the size of the type.

This can be particularly problematic when tyring to determine the size of an array.

### 5.89.1 Parameter Requirements

No Parameter specifications yet!

### 5.89.2 Implementation

Finds calls to the sizeof function. Checks argument for level of pointer (ie pointer to pointer etc...) then checks the number of dereference levels. If these do not match an alert is raised.

### 5.89.3 Non-Compliant Code Example

This non-compliant code example mistakenly calls the sizeof() operator on the variable d_array which is declared as a pointer to double instead of the variable d which is declared as a double.

```
double *d_array;
size_t num_elems;
/* ... */

if (num_elems > SIZE_MAX/sizeof(d_array)){
  /* handle error condition */
}
else {
  d_array = malloc(sizeof(d_array) * num_elems);
}
```

The test of num_elems is to ensure that the multiplication of sizeof(d_array) * num_elems does not result in an integer overflow

For many implementaion, the size of a pointer and the size of double (or other type) is likely to be different. On IA-32 implementations, for example, the sizeof(d_array) is four, while the sizeof(d) is eight. In this case, insufficient space is allocated to contain an array of 100 values of type double.

### 5.89.4 Compliant Solution

Make sure you correctly calculate the size of the element to be contained in the aggregate data structure. The expression sizeof (*d_array) returns the size of the data structure referenced by d_array and not the size of the pointer.

```
double *d_array;
size_t num_elems;
```

```
/* ... */

if (num_elems > SIZE_MAX/sizeof(*d_array)){
  /* handle error condition */
}
else {
  d_array = malloc(sizeof(*d_array) * num_elems);
```

### 5.89.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Write your checker algorithm

### 5.89.6 References

ISO/IEC 9899-1999:TC2 [Viega 05] Section 5.6.8, "Use of sizeof() on a pointer type" [ISO/IEC 9899-1999] Section 6.5.3.4, "The sizeof operator" [Drepper 06] Section 2.1.1, "Respecting Memory Bounds"

## 5.90 Secure Coding : INT06-A. Use strtol() to convert a string token to an integer

Use `strtol()` or a related function to convert a string token to an integer. The `strtol()`, `strtoll()`, `strtoul()`, and `strtoull()` functions convert the initial portion of a string token to `long int`, `long long int`, `unsigned long int`, and `unsigned long long int` representation, respectively. These functions provide more robust error handling than alternative solutions.

### 5.90.1 Non-Compliant Example

This non-compliant code example converts the string token stored in the static array `buff` to a signed integer value using the `atoi()` function.

```
1 int si;
2
3 if (argc > 1) {
4   si = atoi(argv[1]);
5 }
6
```

The `atoi()`, `atol()`, and `atoll()` functions convert the initial portion of a string token to `int`, `long int`, and `long long int` representation, respectively. Except for the behavior on error, they are equivalent to

```
1 atoi: (int)strtol(nptr, (char **)NULL, 10)
2 atol: strtol(nptr, (char **)NULL, 10)
3 atoll: strtoll(nptr, (char **)NULL, 10)
4
```

Unfortunately, `atoi()` and related functions lack a mechanism for reporting errors for invalid values. Specifically, the `atoi()`, `atol()`, and `atoll()` functions:

### 5.90.2 Non-Compliant Example

This non-compliant example uses the `sscanf()` function to convert a string token to an integer. The `sscanf()` function has the same problems as `atoi()`.

```
1 int si;
2
3 if (argc > 1) {
4   sscanf(argv[1], "%d", &si);
5 }
6
```

### 5.90.3 Compliant Solution

This compliant example uses `strtol()` to convert a string token to an integer value and provides error checking to make sure that the value is in the range of `int`.

```
1 long sl;
2 int si;
3 char *end_ptr;
4
5 if (argc > 1) {
6
7   errno = 0;
8
9   sl = strtol(argv[1], &end_ptr, 10);
```

```
10
11   if (ERANGE == errno) {
12     puts("number out of range\n");
13   }
14   else if (sl > INT_MAX) {
15     printf("%ld too large!\n", sl);
16   }
17   else if (sl < INT_MIN) {
18     printf("%ld too small!\n", sl);
19   }
20   else if (end_ptr == argv[1]) {
21     puts("invalid numeric input\n");
22   }
23   else if ('\0' != *end_ptr) {
24     puts("extra characters on input line\n");
25   }
26   else {
27     si = (int)sl;
28   }
29 }
30
```

If you are attempting to convert a string token to a smaller integer type (`int`, `short`, or `signed char`), then you only need test the result against the limits for that type. The tests do nothing if the smaller type happens to have the same size and representation on a particular compiler.

### 5.90.4 Risk Assessment

While it is relatively rare for a violation of this rule to result in a security vulnerability, it could more easily result in loss or misinterpreted data.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---|---|---|---|---|---|
| INT06-A | **2** (medium) | **2** (probable) | **2** (medium) | **P8** | **L2** |

#### Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website .

### 5.90.5 References

[ Klein 02 ] [
ISO/IEC 9899-1999 ] Section 7.20.1.4, "The strtol, strtoll, strtoul, and strtoull functions," Section 7.20.1.2, "The atoi, atol, and atoll functions," and Section 7.19.6.7, "The sscanf function"

## 5.91 Sub Expression Evaluation Order

This checker detects if there exist, within an expression, sub-expressions that update the same variable. As the order of evaluation of such expressions is not guaranteed to be left-to-right, any of the sub-expressions can be taken place first.

### 5.91.1 Parameter Requirements

None.

### 5.91.2 Implementation

This checker uses a simple traversal. For every function call statement, the checker examines 1) whether the function call has sub-expressions that update variables and 2) the variables updated are identical.

### 5.91.3 Non-Compliant Code Example

```
int bar(int a, int b);

void foo()
{
  int i = 0;

  i = bar(++i, ++i); // either ++i could be evaluated first
  i = bar((i=3), (i=4)); // no particular order is guaranteed.
}
```

### 5.91.4 Compliant Solution

```
int bar(int a, int b);

void foo()
{
  int i = 0;

  i = bar(2, 3); // fine
  i = bar((i=2), 3); //fine
}
```

### 5.91.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each node, check if the node is a function call statement.

2. Examine if the function call has sub-expressions that update variables.
3. If yes, examine further if the variables updated are identical.

### 5.91.6 References

The Programming Research Group, High-Integrity C++ Coding Standard Manual, Item 10.3: "Do not assume the order of evaluation of operands in an expression."

## 5.92 Ternary Operator

This checker detects a expression that uses the ternary operator. The rationale for this checker is, according to "High-Integrity C++ Coding Standard Manual", because evaluation of a complex condition is best achieved through explicit conditional statements.

### 5.92.1 Parameter Requirements

None.

### 5.92.2 Implementation

This checker is implemented with a simple traversal. It traverses AST, checks if a statement uses a ternary operator, and reports it if yes.

### 5.92.3 Non-Compliant Code Example

```
void foo()
{
int i = 0;
int j;

(i==3) ? (j=1) : (j=2);
}
```

### 5.92.4 Compliant Solution

```
void foo()
{
int i = 0;
int j;

if(i == 4)
j = 1;
else
j = 2;
}
```

### 5.92.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each node, check if a node represents a ternary operator.
2. If yes, report it

### 5.92.6 References

The Programming Research Group, High-Integrity C++ Coding Standard Manual, Item 10.20: “Do not use the thernary operator(?:) in expressions.”

## 5.93 Time_t Direct Manipulation

"CERT Secure Coding MSC05-A" states

> `time_t` is specified as an "arithmetic type capable of representing times." However, how time is encoded within this arithmetic type is unspecified. Because the encoding is unspecified, there is no safe way to manually perform arithmetic on the type, and, as a result, the values should not be modified directly.

### 5.93.1 Parameter Requirements

This checker takes no parameters and inputs source file.

### 5.93.2 Implementation

This pattern is checked using a simple AST traversal that visits all binary operation nodes. For each binary operation node, if the node is a arithmetic expression then check the type of its left and right hand side operands. If either operand type is `time_t` then flag violation.

### 5.93.3 Non-Compliant Code Example

This code attempts to execute `do_some_work()` multiple times until at least `seconds_to_work` has passed. However, because the encoding is not defined, there is no guarantee that adding `start` to `seconds_to_work` will result adding `seconds_to_work` seconds.

```
#include <time.h>

int do_work(int seconds_to_work)
{
  time_t start;
  start = time( NULL );

  if (start == (time_t)(-1)) {
    /* Handle error */
  }
  while (time(NULL) < start + seconds_to_work)
  {
    //do_some_work();
  }

  return 0;
}
```

### 5.93.4 Compliant Solution

This compliant solution uses `difftime()` to determine the difference between two `time_t` values. `difftime()` returns the number of seconds from the second parameter until the first parameter and returns the result as a `double`.

```
#include <time.h>

int do_work(int seconds_to_work) {
  time_t start;
  time_t current;
  start = time(NULL);
  current = start;

  if (start == (time_t)(-1)) {
    /* Handle error */
  }
  while (difftime(current, start) < seconds_to_work) {
    current = time(NULL);
    if (current == (time_t)(-1)) {
       /* Handle error */
    }
    //do_some_work();
  }

  return 0;
}
```

### 5.93.5 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Perform simple AST traversal on all binary operation nodes.
2. For each binary operation node, if node is arithmetic expression then determine the type of its left and right hand side operands.
3. If either the left or right hand side is type `time_t` then flag violation.
4. Report any violations.

### 5.93.6 References

Secure Coding : MSC05-A. Do not manipulate time_t typed values directly

## 5.94 Unary Minus

The unary minus operator should only be used with signed types, as its use with unsigned types will never result in a negative value. This checker reports any uses of the built-in unary minus operator on an unsigned type.

### 5.94.1 Parameter Requirements

This checker does not require any parameters.

### 5.94.2 Non-Compliant Code Example

```
unsigned int f_noncompliant(unsigned int u)
{
    return -u;
}
```

### 5.94.3 Compliant Solution

```
int f_compliant(int n)
{
    return -n;
}
```

### 5.94.4 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Check the type of the operand of any unary minus expression; emit a diagnostic if it is an unsigned integer type.

### 5.94.5 References

A reference for this rule is: The Programming Research Group: "High-Integrity C++ Coding Standard Manual", Item 10.21: "Apply unary minus to operands of signed type only."

## 5.95 Uninitialized Definition

This test ensures that all variables are initialized at their point of definition. The detector will report all variable declarations without initializer expressions, except for some special cases:

- Variables declared extern: Such declarations refer to variables defined elsewhere, initializers are therefore not required at this point.
- Variables declared static: These variables are automatically initialized to 0 (of the appropriate type) if no explicit initializer is present.
- Class member variables: The class constructor is responsible for initializing such variables.
- Variables of class type: Class objects are default-initialized if no explicit initializer expression is present.
- Variables declared at file ('global') scope: File scope declarations are implicitly static; they can be changed to extern by an explicit modifier. One of the above cases will always apply.

### 5.95.1 Parameter Requirements

This checker does not require any parameters.

### 5.95.2 Non-Compliant Code Example

```
void f_noncompliant()
{
    int x; // not OK, no initializer
}
```

### 5.95.3 Compliant Solution

```
struct foo {
    int member; // OK, class member
};

void f_compliant(int n)
{
    int x = n; // OK, initializer present
    static int y; // OK, static
    struct foo st; // OK, class type (has constructor)
    extern int not_here; // OK, extern
}
```

### 5.95.4 Mitigation Strategies

**Static Analysis**

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each variable declaration without an initializer expression, check the above criteria.
2. If none of the exceptions apply, generate a diagnostic.

### 5.95.5 References

A reference for this rule is: H. Sutter, A. Alexandrescu: "C++ Coding Standards", Item 19: "Always initialize variables".

## 5.96 Upper Range Limit

By always using inclusive lower limits and exclusive upper limits, a whole class of off- by-one errors is eliminated. Furthermore, the following assumptions always apply: 1)the size of the interval equals the difference of the two
2) the limits are equal if the interval is empty
3) the upper limit is never less than the lower limit

Examples: instead of saying x¿=23 and x¡=42, use x¿=23 and x¡43.

### 5.96.1 Parameter Requirements

No parameters required.

### 5.96.2 Implementation

In a fairly straight-forward implementation we search for the greater than or equal to operator.

### 5.96.3 Non-Compliant Code Example

```
int x = 5;
if (x >= 5)
{
x++;
}
% write your non-compliant code example

% write your non-compliant code example
```

### 5.96.4 Compliant Solution

```
int x = 5;
if (x > 4)
{
x++;
}
% write your non-compliant code example

% write your compliant code example
```

### 5.96.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. find greater than or equal to operator
2. raise alert

### 5.96.6 References

Abbreviated Code Inspection Checklist Section 11.1.2, Control Variables"

## 5.97 Variable Name Equals Database Name

For some member function accesses the name of the local variable that gets assigned the result of the function call should have a name equal to the first argument. [ALE3D] E.g:

```
real8 *sx = regM->fieldReal("sx") ;
real8 *syy = regM->fieldReal("sy") ;
```

Where the name of the local variable is not the same as the name in the database for "syy", but it is the same for "sx".

This checker will only report the locations of the assign expressions where this rule is not followed.

### 5.97.1 Parameter Requirements

The checker takes the name of the class and member function that on call should assign it's result to a variable with the same name as the first argument. The arguments are "VariableNameEqualsDatabaseName.ClassName=CLASSNAME" and "VariableNameEqualsDatabaseName.MemberFunctionName=MEMFUNCNAME".

### 5.97.2 Implementation

The checker will look for a function call to the member function within the class that we are looking for. When such a call is found it assumes that the lhs of the last assign expression is the variable access that we are interested in. If the name of that variable does not satisfy the rule we report an error.

### 5.97.3 Non-Compliant Code Example

```
#include <string>
class MixMatmodel{

  public:
    double fieldReal(std::string str){ return 1; };

};

int main(){
    MixMatmodel x;

    int y     = x.fieldReal("test1");
    int z     = x.fieldReal("test1:test2");
    int test2 = x.fieldReal("test1:test2:test3");

};
```

### 5.97.4 Compliant Solution

```
#include <string>
class MixMatmodel{

  public:
    double fieldReal(std::string str){ return 1; };

};

int main(){
    MixMatmodel x;

    int test1 = x.fieldReal("test1");
    int test2 = x.fieldReal("test1:test2");
    int test3 = x.fieldReal("test1:test2:test3");

};
```

### 5.97.5 Mitigation Strategies

#### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. Travese the AST
2. For each call to the member function we are interested in make sure field name is equal to the string provided as an argument.

### 5.97.6 References

# 5.98 Void Star

This checker enforces the guideline of section 87 of *the Elements of C++ Style* (Misfeldt and al., 2004). No public method should be using `void*` type for arguments or return type. If needed, it can be wrapped.

## 5.98.1 Parameter Requirements

There is not parameter requirement.

## 5.98.2 Non-Compliant Code Example

```
class A {
public:
  const void* getData();
};
```

## 5.98.3 Compliant Solution

```
class A {
public:
  const Data getData();
};
```

## 5.98.4 Mitigation Strategies

### Static Analysis

Compliance with this rule can be checked using structural static analysis checkers using the following algorithm:

1. For each public member function declaration, check all argument types and the return type are not `void*`.

## 5.98.5 References

Bumgardner G., Gray A., and Misfeldt T. *The Elements of C++ Style*. Cambridge University Press 2004.

# Chapter 6

# Appendix

## 6.1 Design And Extensibility of Compass Detectors

The design of the detectors is intended to be simple and with little required to be specified to build individual detectors. Of course some detectors may be non-trivial (e.g. null pointer analysis, buffer overflow detectors, etc. (not yet provided in Compass)) the majority are simple. All detectors are meant to be side-effect free and are the subject of separate research to independently provide automated combining (evaluation of multiple patterns in a single AST traversal) and parallelization of the pattern evaluations on the AST.

### 6.1.1 Input Parameter Specification

Parameters to all detectors are specified in an input parameter file (if required). This permits numerous knobs associated with different pattern detectors and separate input files be specified for different software projects.

### 6.1.2 Pattern Detection

Currently it is assumed that patterns will be detected as part of a traversal of the AST. See the ROSE Tutorial for example and general documentation on the different sorts of traversals possible within ROSE.

### 6.1.3 Output Specification

Output of source position information specific to detected patterns are output in GNU standard source position formats. See **http://www.gnu.org/prep/standards/html_node/Errors.html** for more details on this format specification and now it is used by external tools (e.g. emacs, etc.).